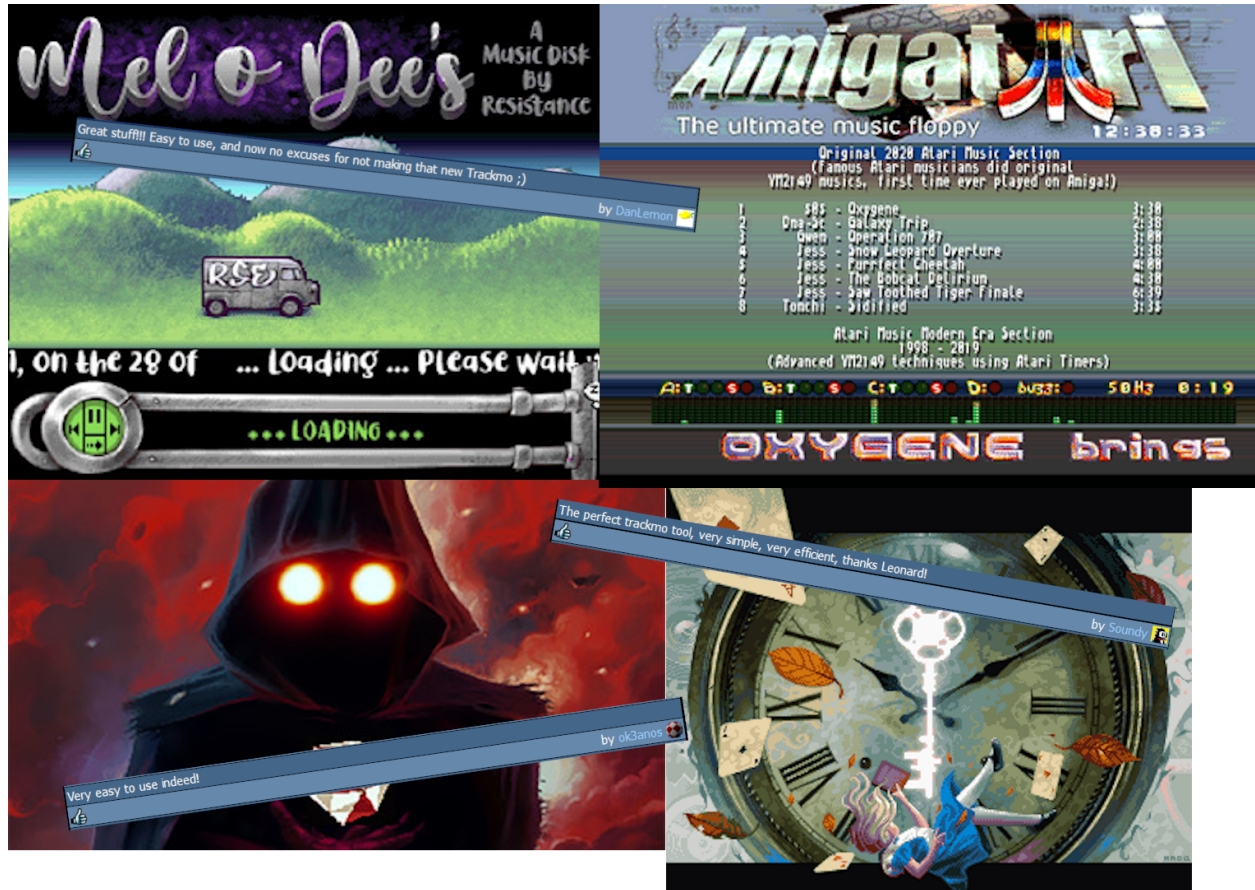


# LDOS

*Leonard's Demo Operating System*



The easiest way to code a trackmo for your Amiga

## TABLE OF CONTENTS

[TABLE OF CONTENTS](#)

[INTRODUCTION](#)

[CREDITS](#)

[FILES TYPES](#)

[GETTING THE FILES YOU NEED](#)

[SCRIPT.TXT](#)

[BUILD.CMD](#)

[API DOCUMENTATION](#)

[LOADING & PLAYING FX](#)

[PLAYING MUSIC](#)

[2 DISKS DEMOS](#)

[RAM MANAGEMENT](#)

[ADDING YOUR OWN ERROR MESSAGES](#)

[FINAL WORDS](#)

## INTRODUCTION

LDOS is a framework to easily build Amiga multi-part demos (often called Trackmos). You can chain several amiga executable effects. LDOS is managing memory allocation, floppy disk loading, data depacking and image disk creation. LDOS also includes an HDD loader to run your demo from harddisk. LDOS toolchain is running on Windows platform.

LDOS is mature, stable, and super easy to use. A growing list of Amiga trackmos have been using LDOS, you can find them here: <https://www.pouet.net/lists.php?which=202>

## CREDITS

- LDOS is written by Arnaud Carré aka Leonard/Oxygene ( [@leonard\\_coder](#) )
- ARJ depackers by Mr Ni! / TOS-crew
- Light Speed Player (LSP) by Leonard/Oxygene (<https://github.com/arnaud-carre/LSPlayer>)
- This doc was written by Soundy (<https://www.pouet.net/user.php?who=13965>)

## FILES TYPES

Here's a summary of the different files types you're going to deal with:

- `.asm` / `.s`  $\Rightarrow$  your assembly source code
- `.bin`  $\Rightarrow$  output of the assembler/linker. These files are generated by `m.cmd` and are used by LDOS to create an ADF file. A `.bin` can be seen as an executable with its code, data and bss sections. In this doc, I often refer to "FX", which are actually a `.bin` file. So ".bin" and "FX" are the same thing, ok?
- `.cfg`  $\Rightarrow$  Compiler & linker configuration file
- `.cmd`  $\Rightarrow$  Windows Batch File, similar to the good old `.bat`
- `.lsmusic` & `.lsbank`  $\Rightarrow$  LSP music player native files formats

## GETTING THE FILES YOU NEED

1. Grab LDOS's latest version here: <https://github.com/arnaud-carre/ldos>
2. In folder `LDOS/demo/` you will find a suggested folder organization:
  - `demo/`
    - `build.cmd`
    - `parcade.mod`
    - `script.txt`
    - `greetings/`
      - `greetings.asm`
      - `m.cmd`
      - `run.cmd`
      - `script.txt`
      - `vc.cfg`
    - `sprites_loader/`
      - `sprite.asm`
      - `m.cmd`
      - `run.cmd`
      - `script.txt`
      - `vc.cfg`

Here's an explanation of the above file organization, remember this is just a **suggestion**, of course you can use your own file structure:

- **demo:** Global folder that contains the whole demo, and specific files to build the full demo:
  - **build.cmd:** used to build the full demo (described below)
  - **parcade.mod:** just the music for your demo (can be several)
  - **script.txt:** lists all the .bin, and the music to copy to the .adf (see below for more explanations about script.txt)
- **greetings, sprite\_loader, ...:** Folders containing the files needed to generate your various FX. Each is composed of:
  - source and data files
  - **m.cmd:** builds and links your code. The output is a “.bin” file.
  - **run.cmd:** creates a .ADF out of the .bin and launches WinUAE.
  - **script.txt:** the list of files to include in the ADF file for this FX (more details about script.txt below).
  - **vc.cfg:** you need it in every FX folder. It contains the arguments (parameters) for the assembler and the linker (vasm, vlink). You can use the one provided here: LDOS\demo\greetings\vc.cfg .

**Note:** If you check m.cmd, you will notice that optimizations are off: “-O0”. This is the default configuration to rebuild LDOS as a library. Now, for your own FX, feel free to enable optimizations and use “-O2”, no problem. Do not modify m.cmd for the LDOS library, do it only for your own FX.

## SCRIPT.TXT

This is a text file containing the names of all the files to include in your Trackmo (basically, the list of files to copy to the ADF disk). 1 file path per line, as easy as that.

- FX Development use case:
  - You need to build an ADF containing only FX1 to test it? Just type a single line in script.txt:

```
FX1.bin
```

See demo file LDOS\demo\greetings\script.txt for example.

Wonder what a .bin is? It's generated by build.cmd, described later.

- Maybe you want to test FX1, but at the end of FX1 you coded a transition to FX2. Wanna test it? Just add FX2.bin in your script. Now your script file in the FX1 folder contains 2 lines:

```
FX1.bin  
FX2.bin
```

- Building the whole trackmo use case:

Now you want to generate an ADF that contains your whole demo. Here's the script.txt you need:

```
music.lsbank  
music.lsmusic  
../FX1/FX1.bin  
../FX2/FX2.bin
```

See demo file `LDOS\demo\script.txt` for example.

The 2 first lines are the files generated by LSP when processing your .MOD music file (explained further in this document).

## BUILD.CMD

This is a text file containing the instructions to build an ADF:

Open demo file `LDOS\demo\build.cmd` for example:

1. calls `m.cmd` for each of the effects of the demo to generate a .bin for each of them
2. converts the music to the LSP format
3. generates the ADF from all the .bin and music files according to `script.txt`
4. invokes WinUAE

## API DOCUMENTATION

All the available functions are listed in `"../..../ldos/kernel.inc"`, and guess what, your code must include `"../..../ldos/kernel.inc"` Then, all functions are invoked the same way:

```
move.l    (LDOS_BASE).w,a6
jsr       FUNCTION(a6)
```

For example:

```
move.l    (LDOS_BASE).w,a6
jsr       LDOS_PRELOAD_NEXT_FX(a6)
```

Note that `../..../ldos/kernel.inc` already provides a basic documentation for the API. You will find some extra information in the chapters below.

## LOADING & PLAYING FX

LDOS will automatically load the first `.BIN` in `script.txt` and run it. When the `.BIN` exits (when the last `RTS` in your code executes), LDOS will load the next `.BIN` and so on... This means there's a loading time between 2 FX (`.bin`), which is bad because you want flawless transitions and a great pacing for your demo. The good news is that you can pre-load the next `.BIN` while the current `.BIN` is playing, using **LDOS\_PRELOAD\_NEXT\_FX**.

**LDOS\_PRELOAD\_NEXT\_FX** will load & depack the next file in `script.txt`. This function is blocking (it returns only when it's 100% done), and of course you don't want to have your current FX paused by preloading. The best practice is to have 2 "threads" in your FX: One is updating your FX for example in the `VBL` interrupt (triggered by your `copperlist`), and the other one is the "main loop" that will preload the next FX and wait for the right time to exit your FX. That way, your FX gets updated at a regular frame-rate and the remaining cycles are spent loading & unpacking the next FX. Once the current FX ends (using `RTS`) the next one is executed immediately.

- Note #1: If the next file in `script.txt` is not a `.bin`, but a music or a data file, then this file will be loaded.
- Note #2: If you don't pre-load, next FX will be loaded & depacked when the previous one ends. So preload is not compulsory, it just avoids a blank screen in your demo while next FX loads (the music keeps on playing though).

- Note #3: Obviously, some extra RAM is needed to preload the next FX, though the bss sections of the next FX will only be allocated at execution.

## PLAYING MUSIC

LDOS features a LSP player. See <https://github.com/arnaud-carre/LSPPlayer> for more info. LDOS\demo\build.cmd performs the conversion from .MOD to LSP format, here's how it works: If for example your music file is "music.mod", then calling "ldos\bin\LSPConvert music.mod -getpos" will generate 2 files:

- music.lsbank (containing the samples, loaded to CHIP RAM by LDOS)
- music.lsmusic (containing the notes, , loaded to ANY RAM by LDOS)

Include these files in your script.txt, and use the following LDOS music API:

- **LDOS\_MUSIC\_START:** Will start playing the latest loaded music. 2 ways to load a music:
  - Either the music files are the 2 first files in script.txt and the music will be ready to play when your first FX starts (see LDOS\demo\script.txt for an example).
  - Either the music is NOT at the beginning of script.txt, and it needs to be preloaded using **LDOS\_PRELOAD\_NEXT\_FX**. This method is generally used when you have several musics in a single demo.
- **LDOS\_MUSIC\_GET\_TICK:** Get LSP music frame tick. Use this counter if you want to sync gfx with music (output: d0.l = music frame tick). The return value is actually the CIA counter ticks, and it totally depends on the music's tempo. There is no trivial way to convert the tick counter into patter/pos/row in the original .MOD file.
- **LDOS\_MUSIC\_GET\_SEQ\_POS:** Returns the current MOD music sequence position (output: d0.w = music sequence position). Basically, this number increases every time a new pattern starts. Note this is not the pattern number, but the index in the pattern sequence. In other words, if your song plays 4 times pattern 0, LDOS\_MUSIC\_GET\_SEQ\_POS will return 1,2,3,4 and not 0,0,0,0.
- **LDOS\_MUSIC\_STOP:** Stops the current music and frees the associated RAM.
- ***What if I want to play a music while another one is loading?***  
Imagine you want to load your main music while an intro music is playing.

Unfortunately, **LDOS supports only a single music at a time** (that means you can't play a music while you are loading another one). The trick is to hide your intro music and the LSP player within your intro FX: Include your intro music (incbin) in your intro fx (.bin), and play it using LSP replay routine (include LSP source code in your intro's source code and compile it with your fx in the same .bin). While your intro is playing, you can load your main music using script.txt.

## 2 DISKS DEMOS

Here's how to handle multiple disks demos.

- First, you need 2 script.txt files in your demo/ folder: one for each disk, listing which .bin and which musics go to which disk.
- Then, here's how you check for disk swap:

```
.retry:
move.l      (LDOS_BASE).w,a6
move.w      #5*50,d0 ; wait 5 seconds in case of HDD version
jsr         LDOS_IS_DISK2_INSERTED(a6)
cmp.b       #$ff,d0
bne.b       .retry
; success
move.l      (LDOS_BASE).w,a6
jsr         LDOS_PRELOAD_NEXT_FX(a6)
```

Obviously, this works for any amount of disks (not only 2).

## RAM MANAGEMENT

If you're lucky enough, you won't have to worry about it. LDOS will allocate your code, data and bss sections for each .BIN and handle RAM gracefully for you, free of fragmentation. For example, [The Fall](#) and [De Profundis](#) were developed without ever wondering about RAM management. LDOS has 2 RAM buffers: 1 for CHIP RAM, and 1 for ANY RAM, that is also referred as FAKE RAM (though the FAKE RAM buffer could be CHIP on a A600 for example). Each RAM buffer is split in 4kb pages (also called buckets or slots).



- ***How do I allocate RAM in my FX?***

- Are you sure? I mean, are you really really sure? If you can go with BSS sections, always prefer BSS to malloc. DATA sections are fine too, but obviously make your .bin files bigger than a BSS, and also take more RAM to preload your FX. LDOS will allocate your sections for you when your FX starts and free them when your FX ends. Now if for some reason you need more RAM during your FX and you need to free that RAM before you preload the next FX, or if you need to alloc some RAM that will be accessible to other FX, keep on reading ;)
- Keep in mind that LDOS' memory manager rounds everything to the nearest 4Kb boundary. So if you want to alloc two buffers of 1024 bytes each, consider allocating a single 2048 bytes buffer as a single malloc (which will allocate one 4kb page) instead of two 1024 bytes buffers (which would allocate two 4kb pages, so 8k when you only need 2k).
- You may want to allocate buffers in a FX that will be used by other FX. In that case, see: `LDOS_PERSISTENT_CHIP_ALLOC / LDOS_PERSISTENT_CHIP_GET / LDOS_PERSISTENT_CHIP_TRASH`

Be aware that allocating persistent RAM may lead to memory fragmentation. To avoid this, alloc your persistent RAM as early as possible in your demo. Don't hesitate to start your demo (in script.txt) with a tiny .bin that does nothing but allocate the shared RAM. This was used for example in "The Nature Of Magic" by NGC : Persistent CHIP RAM is allocated at the beginning of the demo to store a default copperlist installed at the end of each FX. This way, the background color remains the same between 2 FX.

- *I ran out of RAM, how can I investigate it?*

So you reached the “MALLOC ERROR!” assert screen. I sympathize, really. You have some tools to investigate though. You may have different components eating your RAM:

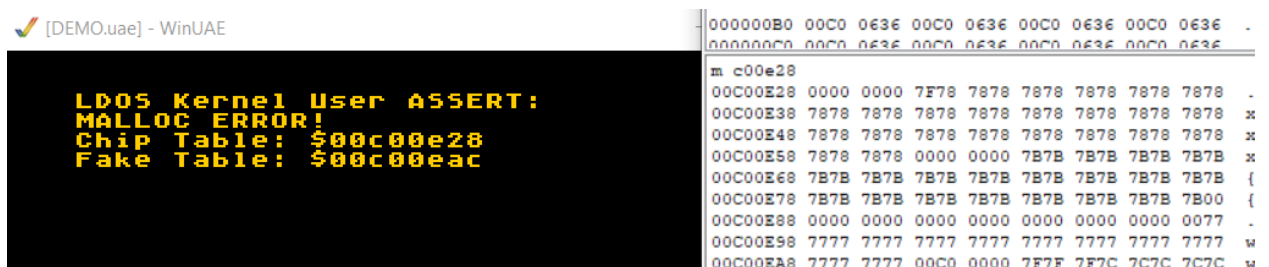
- The music (.lsmusic in ANY RAM, .lsbank in CHIP RAM)
- The current FX various sections (code, data, bss)
- The pre-cached next FX (if you used LDOS\_PRELOAD\_NEXT\_FX)
- Some persistent RAM (if you used LDOS\_PERSISTENT\_xxx)

First, check the info given by LDOS when packing your .bin to the ADF:

```
[0]:$000000 [Raw] Packing 292-> 292 (100%) Chip: 0KiB Fake: 0KiB (boot.bin)
[0]:$000124 [Raw] Packing 9454-> 7046 ( 74%) Chip: 0KiB Fake: 10KiB (kernel.bin)
[0]:$001caa [Exe] Packing 53636-> 20108 ( 37%) Chip:188KiB Fake: 44KiB (streamers.bin)
```

Maybe one of your .bin is taking too much RAM, or cannot fit in RAM while another huge .bin is being preloaded. Remember though that the BSS sections are not allocated during preload, but only at execution. This means that only 1 FX at a time has its bss allocated. Still, if for example you have a big music, you are playing a FX that takes a lot of bss\_chip, and the next FX has a big chip section (e.g. lots of images as incbin), you may run out of CHIP RAM.

Now, you may need more info, maybe you need to know if you fragmented the RAM abusing LDOS\_PERSISTENT\_xxx commands, or you just want to know exactly what every 4kb bucket of RAM is used for. We got you: The “MALLOC ERROR!” screen gives you the “Chip Table” and “Fake Table” addresses that can be investigated in your WinUAE debugger: Each table starts with the address of the RAM buffer (4 bytes), then the 128 following bytes give the status of each 4KB page within the buffer. 0 means the page is free, so by checking the tables, you can know if you ran out of free or fake memory. For example, here’s a rather full CHIPmem table (starts at address 0, then all pages are allocated):



The screenshot shows the WinUAE debugger interface. On the left, a black window displays the text: "LDOS Kernel User ASSERT: MALLOC ERROR! Chip Table: \$00c00e28 Fake Table: \$00c00eac". On the right, a memory dump is visible, showing hexadecimal values and their corresponding ASCII representations. The dump starts with "000000B0 00C0 063E 00C0 063E 00C0 063E 00C0 063E" and continues with a series of "7F78" values, indicating that the memory is mostly filled with the ASCII string "c00e28".

But wait, there is more! Got to <http://deadliners.net/LDOSRAM/> and here's what you'll get:

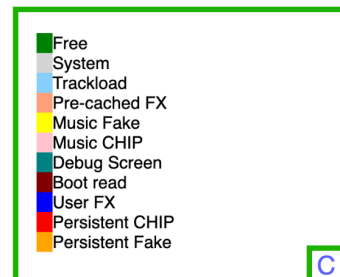
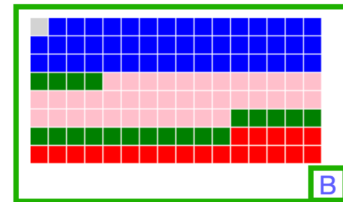
WinUAE Dump:

```

00C00E48 0000 0000 7F78 7878 7878 7878 7878 .....
00C00E58 7878 7878 7878 7878 7878 7878 7878 .....
00C00E68 7878 7878 7878 7878 7878 7878 7878 .....
00C00E78 7878 7878 0000 0000 7B7B 7B7B 7B7B .....
00C00E88 7B7B 7B7B 7B7B 7B7B 7B7B 7B7B 7B7B .....
00C00E98 7B7B 7B7B 7B7B 7B7B 7B7B 7B7B 7B00 .....
00C00EA8 0000 0000 0000 0000 0000 0000 0077 .....
00C00EB8 7777 7777 7777 7777 7777 7777 7777 .....
00C00EC8 7777 7777 00C0 0000 7F7F 7F7C 7C7C 7C7C .....

```

LDOS RAM



Memory:

```

FREE: 81920 bytes (80 Kb)
SYSTEM: 4096 bytes (4 Kb)
TRACKLOAD: 0 bytes (0 Kb)
PRECACHED_FX: 0 bytes (0 Kb)
LSP FAKE: 0 bytes (0 Kb)
LSP CHIP: 159744 bytes (156 Kb)
DEBUG_SCREEN: 0 bytes (0 Kb)
BOOTREAD: 0 bytes (0 Kb)
USER_FX: 192512 bytes (188 Kb)
PERSISTENT_CHIP: 86016 bytes (84 Kb)
PERSISTENT_FAKE: 0 bytes (0 Kb)

```

Free Zones:

```

start slot: 48 - size: 16384
start slot: 91 - size: 65536
largest free zone: 65536 bytes (64 Kb)

```

**Zone A:** Paste here what WinUAE's memory viewer gave you. You need to paste the dump for 132 bytes starting from the table address given by the memory assert. Why 132 bytes? 4 bytes are for the memory buffer's start address, then the remaining 128 bytes store the status of each of the 128 buckets ( $128 * 4\text{kb} = 512\text{Kb}$ , we're good!). If you think it's a hassle to copy/paste the data for exactly 132 bytes, you're right! That's why you may paste more than that, the unneeded extra bytes will just be ignored. Check the screenshot on the previous page: in WinUAE's debugger, we typed "m c00e28" to dump memory from this address and see what's in the CHIP memory table. Just paste what WinUAE dumped in the "WinUAE Dump" window.

Note: Did you notice? In the above screenshot, the dumped memory starts at c00e48 and not c00e28. It's only because the screenshots were not taken during the same debugging session. Sorry for the incoherence. Obviously, the start address should be the one given by LDOS in the assert screen.

**Zone B:** This shows you how each 4kb RAM bucket is used. The color codes are shown in **Zone C**.

**Zone D** shows more info (sums up all buckets used for the same RAM usage). In this screenshot, no FAKE/ANY RAM is used. The reason is simple: We dumped the address of the CHIP RAM table!

**Zone E** shows all the free RAM zones.

## ADDING YOUR OWN ERROR MESSAGES

LDOS crash screens will give you an explanation of why it crashed. But you can use this feature to implement your own assert/error system. To see examples, look for “trap #0” instructions in LDOS source files. For example:

```
InvalidParam:
    move.w    d0,-(a7)        ; push param to display on stack
    movea.l   a7,a1
    lea       .txt(pc),a0     ; error message
    trap      #0              ; trigger assert
.txt:
    dc.b      'D0.W = %w Invalid param!',0
    even
```

Another example:

```
mallocError:
    lea       .txt(pc),a0     ; error string
    pea       fastMemTable(pc) ; 2nd displayed argument
    pea       chipMemTable(pc) ; 1st displayed argument
    movea.l   a7,a1
    trap      #0              ; assert
.txt:
    dc.b      'MALLOC ERROR!',10 ; “,10” goes to the next line
    dc.b      'Chip Table: %l',10 ; %l ⇒ chipMemTable(pc)
    dc.b      'Fake Table: %l',10 ; %l ⇒ fastMemTable(pc)
```

```
dc.b      0      ; classic 0 terminated string
even      ; better safe than sorry ;)
```

## FINAL WORDS

LDOS is an incredibly easy and efficient tool, it features:

- Easy trackmo ADF creation
- Best in class packing / unpacking routines
- Best in class music player

All of this has been tested for years on award winning demos. You're in good hands. Leonard is adding new features on a regular basis, so stay tuned. Also, and most important, you're not alone! Leonard is super reactive to help (just add a comment on [LDOS's Pouet page](#)), and if you have any questions reading this doc or concerning my experience using (and customizing) LDOS, I'm also happy to help! Just write to soundydeadliners at gmail.com