JOTD HD Install Collection

COLLABORATORS			
	TITLE : JOTD HD Install Collection		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 8, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 JOTD HD Install Collection **1**

1.1 JOTD HD Startup code autodocs . . . 1
1.2 Memory setup . . . 1
1.3 Calling the user code . . . 2
1.4 Useful macros . . . 3
1.5 STORE_REGS-RESTORE_REGS . . . 5
1.6 Console output . . . 5
1.7 PatchMacros . . . 6
1.8 CallMacros . . . 6
1.9 Compelling Macros . . . 7
1.10 MiscMacros . . . 8
1.11 Relocatable Memory operations . . . 9
1.12 Registers: CAUTION! . . . 10
1.13 Why decrunch routines? . . . 10
1.14 Functions list . . . 11
1.15 JoypadState() . . . 15
1.16 GetUserData() . . . 16
1.17 GetUserFlags() . . . 16
1.18 SetLocalVarZone() . . . 16
1.19 SetFilesPath() . . . 17
1.20 UseHarryOSEmu() . . . 17
1.21 DisableChipmemGap() . . . 18
1.22 ReadUserDir() . . . 18
1.23 SaveOSData() . . . 19
1.24 Reboot() . . . 20
1.25 AllocExtMem(), Alloc24BitMem() . . . 20
1.26 OpenFakeExec() . . . 21
1.27 FreezeAll() . . . 23
1.28 LoadDisks() . . . 23
1.29 LoadDisksIndex() . . . 23

1.30 LoadDiskFromName() . . . 23
1.31 LoadSmallFiles() . . . 24
1.32 LoadFiles() . . . 24
1.33 GetFileLength() . . . 24
1.34 TestFile() . . . 25
1.35 TestFileAbs() . . . 25
1.36 LoadRNCFile() . . . 25
1.37 TransfRoutines() . . . 26
1.38 BlockFastMem() . . . 26
1.39 Degrade() . . . 26
1.40 Enhance() . . . 27
1.41 DegradeGfx() . . . 27
1.42 DegradeCpu() . . . 27
1.43 Display() . . . 28
1.44 CloseAllQuiet() . . . 28
1.45 CloseAll() . . . 28
1.46 FlushCachesSys() . . . 29
1.47 GetMemFlag() . . . 29
1.48 TestxMbChip() . . . 29
1.49 CheckAGA() . . . 30
1.50 GetSR() . . . 30
1.51 GetAttnFlags() . . . 30
1.52 WriteFileHD() . . . 30
1.53 WriteUserFileHD() . . . 31
1.54 ReadUserFileHD() . . . 31
1.55 ReadFile() . . . 31
1.56 ReadFilePart() . . . 32
1.57 ReadFilePartHD() . . . 32
1.58 ReadFileHD() . . . 32
1.59 DeleteFileHD() . . . 33
1.60 DeleteUserFileHD() . . . 34
1.61 InGameOSCall() . . . 34
1.62 PatchExceptions() . . . 36
1.63 FlushCachesHard() . . . 36
1.64 GoECS() . . . 37
1.65 ResetDisplay() . . . 37
1.66 ResetSprites() . . . 37
1.67 KickVerTest() . . . 38
1.68 Kick37Test() . . . 38

1.69 InitTrackDisk() . . . 38
1.70 TrackLoad . . . 38
1.71 SetTDUnit() . . . 39
1.72 ReadFileFast() . . . 39
1.73 ReadRobSectors() . . . 41
1.74 ReadDiskPart() . . . 41
1.75 StrcpyAsm() . . . 42
1.76 StrcmpAsm() . . . 42
1.77 StrlenAsm() . . . 42
1.78 ToUpperAsm() . . . 42
1.79 HexReplaceLong() . . . 42
1.80 HexReplaceWord() . . . 43
1.81 SetFakeFunction() . . . 43
1.82 CRC16() . . . 44
1.83 HexToString() . . . 44
1.84 Save & Restore hardware registers . . . 45
1.85 RNCLength() . . . 45
1.86 RNCDecrunch() . . . 45
1.87 RNCDecrunchEncrypted() . . . 46
1.88 ATNDecrunch() . . . 46
1.89 PPDecrunch() . . . 47
1.90 FungusDecrunch() . . . 47
1.91 BlackScreen() . . . 47
1.92 EnterDebugger() . . . 47
1.93 SetTraceVector() . . . 48
1.94 WaitMouse() . . . 48
1.95 WaitMouseInterrupt() . . . 48
1.96 InGameExit() . . . 49
1.97 IsRegistered() . . . 49
1.98 SetExitRoutine() . . . 49
1.99 InGameIconify() . . . 50
1.100PatchMoveCList_Abs() . . . 52
1.101PatchMoveCList_Ind() . . . 53
1.102PatchMoveCList_Idx() . . . 53
1.103PatchMoveBlit_Idx() . . . 54
1.104StoreCopperPointer() . . . 54
1.105TellCopperPointer() . . . 55
1.106BeamDelay() . . . 55
1.107WaitBlit() . . . 56

1.108 SetQuitKey() 56
1.109 SetIconifyKey() 56
1.110 LogPatch() 57
1.111 InitLogPatch() 57
1.112 The printlog tool 57
1.113 Contact us 58

Chapter 1

JOTD HD Install Collection

1.1 JOTD HD Startup code autodocs

```
                ***************************************************************************

                          JOTD Startup V3.1
                © Copyright 1995-99 Jean-François Fabre/Ralf Huvendiek

*****************************************************************************
```


Memory setup

Starting your code

Why decrunch routines ?

Macros

Library Functions

The printlog tool

Contact us

1.2 Memory setup

To understand some choices and notions, you have to know how the memory
is organized by my loading routines:

On a fastmem amiga:

V39:

High Fast: User object, General reloc routines, disks images or RAM files,

```
           OS chip mirror, extension mem (for the 1MB games) (if any)
Fast: The JOTDStartup program routines not relocated.
Low Chip: The game (can take 512K, 1024K or 2048K)

Older than V39:

Fast: User object, General reloc routines, disks images or RAM files,
           OS chip mirror, extension mem (for the 1MB games) (if any),
           the HD load program
Low Chip: The game (can take 512K, 1024K or 2048K)


On a chip only amiga (V39):

High Chip: User object, General reloc routines, disks images or RAM files,
           OS chip mirror, extension mem (for the 1MB games) (if any)
Chip: The game, The JOTDStartup program routines not relocated.

Obviously, while the game is running, you cannot call non-relocated routines
(the ones callable with JSRABS), or else it will crash. You won't need
those routines anyway during the game (you can use JSRABS routines in
a code called using InGameOSCall)

If the game needs more than 512K to run, you'll have to find where the expansion
memory detection is done to replace it by a pointer on the extension mem you  ↩
    allocated
with AllocExtMem(), else the game can crash (older than V39)

The NOFAST tooltype will try to allocate chipmem for AllocExtMem unless you've got
no chipmem above $80000 (the assumed chip size before you make a SaveOSData())
In this case it will return 0 (it's useless to allocate mem that will be  ↩
    overwritten by
the game)

Note: If you succeed in tracking the game memory 'allocation', you don't need  ↩
    relocatable
routines, and nothing will overwrite your code,
but it depends on which level of quality you want when you program your installers ↩
    ...
And this is also compulsory if you want to install a clean quit option!

In the case you would like to leave some OS code done by the game, JST supports  ↩
    ExecBase
emulation since V0.7. Some calls are supported, others are redefinable by the user ↩
    ,
(see OpenFakeExec() function)
```

1.3 Calling the user code

```
Your relocatable object file will be loaded by JST and it will be checked
against relocatable symbols. For instance, you cannot do things like:

  move.l  #20,var

because this instruction needs relocation, and we can't afford it since
```

```
JST is not yet able to do this. Instead you can use one of the reloc macros.
This instruction will become:

  RELOC_MOVEL #20,var

JST passes some arguments in registers:

D0.L: 0 if no trainer detected via tooltypes/args, -1 if trainer requested
D1.L: 0 if OS swap is allowed, -1 if not (see InGameOsCall())
D2.L: -1 if JOYPAD is set in the tooltypes, 0 else
D3.L: -1 if HDLOAD is set in the tooltypes, 0 else
D4.L: -1 if BUTTONWAIT is set in the tooltypes, 0 else
D5.L: -1 if LOWMEM is set in the tooltypes, 0 else

The others (function tables) are passed in the HD_PARAMS structure.
Use the macros, and don't fool with the variables in HD_PARAMS.
```

1.4 Useful macros

```
                Some of these macros are compulsory for a good compilation and ↩
                   linking
of the user program. They are compatible with phxass and should cause no
problem with other assemblers.



                STORE_REGS

                RESTORE_REGS

                Mac_printf

                PUTS

                NEWLINE

                GETUSRADDR

                GETGENADDR

                PATCHUSRJMP

                PATCHUSRJSR

                PATCHABSJMP

                PATCHABSJSR

                PATCHGENJMP

                PATCHGENJSR

                JSRGEN

                JSRGEN_FREEZE
```

```
JMPGEN

JSRABS

JMPABS

TESTFILE

WAIT_BLIT

BEAM_DELAY

GETLVO

WAIT_JOY

WAIT_LMB

HDPARAMS

SAVEOS_DATA

SET_VARZONE

RELOC_MOVEL

RELOC_MOVEW

RELOC_MOVEB

RELOC_CLRL

RELOC_CLRW

RELOC_CLRB

RELOC_STL

RELOC_STW

RELOC_STB

RELOC_TSTL

RELOC_TSTW

RELOC_TSTB

RELOC_SUBL

RELOC_SUBW

RELOC_SUBB

RELOC_ADDL
```

```
                RELOC_ADDW

                RELOC_ADDB

                Using registers
```

1.5 STORE_REGS-RESTORE_REGS

```
STORE_REGS
Save all the registers to stack (D0 to A6)

RESTORE_REGS
Restore all the registers from stack (D0 to A6)

No parametrers expected.
```

1.6 Console output

```
                Mac_printf:

Prints messages with an easy syntax in an assembly program. Avoids label  ↩
   definition,
and all the problems going with it.

  Mac_printf  "Hello world"
  Mac_printf  "You are ",jkjdkfj
  Mac_printf  "a hd-installer maniac"

will display on the console:

Hello world
You are a hd-installer maniac

If you use 2 arguments in the macro, there will be no linefeed.


PUTS:

Puts a string on the screen
Argument: string pointer.

NEWLINE:

Just skips a line. No argument needed.


Note: Those macros use the
                Display()
                 routine.
```

1.7 PatchMacros

```
GETUSRADDR
Returns in D0 the relocated address of a routine in the user code.
This avoids a lea addr(pc),A0 then move.l A0,D0, but is not really useful anymore ↩
    in
the JST environnement.

GETGENADDR
Returns (in D0) the address of a relocated general purpose routine.

Example:

  lea $1456(A1),A0
  GETGENADDR  ReadRobSectors
  move.w  #$4EF9,(A0)+
  move.l  D0,(A0)+

PATCHUSRJMP
Sets a JMP to a given user relocatable address. The patched location is absolute.
It may be used for chipmem, but not for extension mem, since you cannot know the
absolute address (you have to calculate it). Do it by hand in this case.

PATCHUSRJSR
Same thing, but with a JSR

PATCHABSJMP
PATCHABSJSR
Obsolete. Same thing as PATCHUSRJMP/JSR

PATCHGENJMP
Sets a JMP in the general purpose relocated area (i.e. the functions of this
library callable with JSRGEN)

PATCHGENJSR
Same thing, for JSRs
```

1.8 CallMacros

```
JSRGEN
JSRGEN_FREEZE
JMPGEN
Calls a relocated general purpose function.

E.g:  JSRGEN  TrackLoad

Those macros use A5 to get the function address. This register is restored.

JSRGEN_FREEZE can be useful if A5 (and also other registers used by the JST ↩
    function)
is used during interrupts.
This macro freezes all the interrupts by setting SR to $2700 (means that
you've got to be in superuser mode to use it), executes
```

```
the function, then restores SR. I had to use it in Menace and Lotus Turbo  ↩
    Challenge.

JSRABS
JMPABS
Calls a absolute general purpose subroutine

E.g:  JSRABS  LoadDisks
```

1.9 Compelling Macros

```
                Those macros are used in (almost) every loader.

GO_SUPERVISOR
Calls the Supervisor exec routine and stores the userstack in a variable.No  ↩
    argument needed.
Since v1.1d GO_SUPERVISOR can be called twice safely (there's a check for user  ↩
    mode)

SAVE_OSDATA
Calls
                SaveOSData()
                 with the chipmem you want to be saved.
If you want to save 512K chipmem, type
  SAVE_OSDATA $80000

Passing $200000 to the function will allocate as much as chipmem possible, and
mirror only the other parts (0-low boundary and high boundary-$200000)

If you want to save all the 2MB of chipmem:

  SAVE_OSDATA $200000

This will allocate a big block of chipmem, and this block won't have to be  ↩
    mirrored.

Passing 2 arguments to SAVE_OSDATA will activate autoquit key feature.
The 2nd argument is the autoquit key that you want to use (raw keycode)

Passing 3 arguments to SAVE_OSDATA will allow you to specify some custom user code
you want called on exit (just like in SetExitRoutine())
The 3rd argument is the address of the custom routine.

HD_PARAMS
A compulsory macro which declares some variables, and also set tags for JST to  ↩
    read
and write data and also recognize the loader.

  HD_PARAMS "game.d",filesize,number of disks

The filesize can be set to STD_DISK_SIZE (901120) for normal trackload copiable  ↩
    disks.
See the examples.
```

```
In diskfile mode, this call makes JST expect the diskfiles to be named game.d1,  ↩
    game.d2, etc...
In files mode, the first argument allows to specify the subdir where the data  ↩
    files are
to be located. (But you can also use SetFilesPath() to set the subdir too)

For a loader using files located in the same directory as the loader:

  HD_PARAMS "",0,0



WARNING: This macro has 3 arguments. It's not possible to choose the window name  ↩
    any more.
An error message will show up when assembling if you put the wrong number of  ↩
    arguments.

SET_VARZONE:

A macro useful in the case you want to add snapshot option to your loader and you' ↩
    ve got local
variables that you need to save in the loader (e.g. current disk unit, some flags ↩
    ...)

Put it everywhere you want. The best thing is to put it right after HD_PARAMS  ↩
    macro.
The code will be executed even before your loader code.

This macro consists in a simple JSRABS SetLocalVarZone, with in A0 the start  ↩
    location of the
variables (argument 1 of the macro) and in A1 the end location of the variables ( ↩
    argument 2
of the macro).

All registers are preserved.

If you pass invalid values to this routine (e.g start after end) JST will tell it  ↩
    and quit.
```

1.10 MiscMacros

```
                WAIT_LMB:

A simple left mouse button pause (port 0).

WAIT_JOY:

A simple joystick button pause (port 1).


TESTFILE:

Calls the
                TestFile()
```

```
                function, with relocatable argument.

  TESTFILE  fname
  tst.l D0
  bne FileError

Actually this routine takes D0 as input and that's a bit annoying since in a  ↩
    relocatable
environment we have to lea (pc) in an address register then move to D0 and call  ↩
    the routine.
I don't want to change the way the routine works (backwards compatibility)
but this little macro makes life easier.


GETLVO:

A simple macro to get LVO offset in D0.

Example:  GETLVO   AvailMem   will return negative offset of AvailMem

Intended to use with SetFakeFunction()


WAIT_BLIT

A macro to wait for the blitter operation to end (equivalent to WaitBlit
but does not use any registers, useful for games which use/modify registers
in interrupts)

BEAM_DELAY

A macro to wait using the vertical beam. Takes one argument.

  BEAM_DELAY  1

will wait approx. 20 ms (PAL)
```

1.11 Relocatable Memory operations

```
RELOC_MOVEL/W/B:

A macro which allows to move some data in an address specified by a label, but in  ↩
    a relocatable
way:

  move.l  D0,label  ; is not relocatable and JST will refuse it

  move.l  A6,-(sp)
  lea label(pc),A6
  move.l  D0,(A6)   ; is relocatable. but a pain to write
  move.l  (sp)+,A6

That's what RELOC_MOVEL does.
```

```
For Word and Byte format, similar macros known as RELOC_MOVEW and RELOC_MOVEB  ↩
    exist.

RELOC_TSTL/W/B:

With the 68020+ instruction set, it's possible to test a label in a relocatable  ↩
    way:

  tst.l label(pc)

But with the 68000 this is not possible. So I made a useful macro:

  RELOC_TSTL  label


RELOC_CLRL/W/B:

Makes a RELOC_MOVE/W/B #0 to location

RELOC_ADDL/W/B:
RELOC_SUBL/W/B:

relocatable add/sub
```

1.12 Registers: CAUTION!

```
Be careful not to use registers A5 and A6 with some macros, because they
often use those registers: e.g:

        PATCHUSRJMP     ($30,A5),PatchLoader

will crash because PATCHUSRJMP uses A5. This is subject to change, though.
To change working registers (A5,A6) used in the macros to lets say (A3,A4),
just include the following lines BEFORE the jst.i include.

REDEFINED_REGISTERS = 1 ; to tell jst.i that we've redefined the registers
Ax EQUR A3    ; work register #1
Ay EQUR A4    ; work register #2


Or manually:

        move.l   A0,-(sp)
        move.l   A5,A0
        PATCHUSRJMP      ($30,A0),PatchLoader
        move.l   (sp)+,A0
```

1.13 Why decrunch routines?

```
Why decrunch routines in this package?

2 main reasons:

- If you patch the decrunch entry point in chipmem by a JMP,
  the data will decrunch MUCH faster -> better loader.

- You can also patch the decrunch entry point by a JSR and then do some
  patch stuff easily as the code is now unpacked. Then RTS.
```

1.14 Functions list

```
                Absolute callable routine (JSRABS only)


                CloseAll()

                CloseAllQuiet()

                LoadDisks()

                LoadDiskFromName()

                LoadDisksIndex()

                LoadFiles()

                LoadSmallFiles()

                LoadRNCFile()

                Kick37Test()

                KickVerTest()

                GetMemFlag()

                TransfRoutines()

                SaveOSData()

                FreezeAll()

                Display()

                AllocExtMem()

                Test1MBChip()

                Test2MBChip()

                Reboot()
```

```
                Enhance()

                Degrade()

                DegradeCpu()

                DegradeGfx()

                FlushCachesSys()

                TestFile()

                TestFileAbs()

                OpenFakeExec()

                SetFilesPath()

                SetLocalVarZone()

                InitLogPatch()

                UseHarryOSEmu()

                DisableChipmemGap()
                Relocate callable routines (only with JSRGEN)

Low level calls

                GetSR()

                GoECS()

                SaveCustomRegs()

                RestoreCustomRegs()

                SaveCIARegs()

                RestoreCIARegs()

                InGameOSCall()

                InGameIconify()

                ResetDisplay()

                PatchExceptions()

                InGameExit()

                GetAttnFlags()

                ResetSprites()

                BlackScreen()
```

```
BlockFastMem()

WaitBlit()

BeamDelay()

PatchMoveBlit_Idx()

LogPatch()

SetQuitKey()

SetIconifyKey()
Iconify related routines (new since JST V1.0)


PatchMoveCList_Abs()

PatchMoveCList_Ind()

PatchMoveCList_Idx()

InGameIconify()

StoreCopperPointer()

TellCopperPointer()
HD/floppy emulation related routines


InitTrackDisk()

TrackLoad()

SetTDUnit()

ReadRobSectors()

ReadDiskPart()
SetDisk()

ReadFile()

ReadFileFast()

ReadFileHD()

ReadFilePart()

ReadFilePartHD()

WriteFileHD()

DeleteFileHD()

ReadUserFileHD()
```

```
                WriteUserFileHD()

                DeleteUserFileHD()
                GetDiskPointer()

                ReadUserDir()

                SetFakeFunction()

                GetUserData()

                GetUserFlags()
                ReadFileFromImage()
WriteFilePartHD()

Unpack routines


                RNCDecrunch()

                RNCDecrunchEncrypted()

                RNCLength()

                ATNDecrunch()

                PPDecrunch()
                FireDecrunch()

                FlushCachesHard()
                Utility


                StrcmpAsm()
                StrncmpAsm()

                StrcpyAsm()
                StrncpyAsm()

                StrlenAsm()

                ToUpperAsm()

                HexToString()

                CRC16()

                CheckAGA()

                HexReplaceLong()

                HexReplaceWord()

                JoypadState()

                IsRegistered()
```

```
Debug/Development


WaitMouse()

WaitMouseInterrupt()

EnterDebugger()

SetTraceVector()
```

1.15 JoypadState()

```
ULONG JoypadState(port)
 D0                D0

This routine allows to check CD32 joypad buttons. It obviously does not use
lowlevel.library and is buggy unless on a CD32 (why???). It's ripped from
James Pond 3 routine (I was not the first since the ReadJoypad.s from aminet
was also ripped from this one).

port can be 0 (mouse port) or 1 (joystick port).

The routine returns bits that you can test. The bits are defined in gp_macros.i

  moveq #0,D0
  JSRGEN  JoypadState
  btst  #AFB_START  ; start/pause button
  beq nostart$
  ...

On a normal amiga, you've got to go in a direction to make this routine work.
I don't know why, but it works perfectly on a CD32.
Also, you can't press fire (red button) and the other buttons together, or only ↩
    fire
will be detected.

You have to wait for a vertical blank to occur before you can use this function.
I suggest that you put it in the level 3 interrupt, just before acknowledge
(which is a move in $DFF09C) and set the flags here. Then another routine will
check those flags and emulate game keys such as pause, jump/fire, etc... it's
up to you :)

NOTE:

If you only want to check the second button (blue, or Sega joystick 2nd button),
you can also read $DFF016 and check for bit 14 low, still after a VBLANK.
Don't forget to reset the potgo in that case by

  move.w  #$CC01,($DFF034)

See also:
```

1.16 GetUserData()

```
APTR GetUserData(void)
 A0

This routine returns in A0 a pointer to the NULL terminated string passed by
the USERDATA argument or tooltype. It points to an empty string if USERDATA
was not filled. It's useful to set a swith other than the user ones already ↩
    present
such as TRAINER, JOYPAD... and some info can be passed (it's not only a switch)

I used it in the Awesome JST ripper to specify the disk, side and drive unit.

See also: GetUserFlags()
```

1.17 GetUserFlags()

```
ULONG GetUserFlags(void) - returns loader set options
 D0

This routine returns in D0 a combination of the option the user could have set.
Not all the options are present here, only some switches which are:

TRAINER
HDLOAD
LOWMEM
NTSC
NOOSSWAP
JOYPAD

To test if an option is active, just do a btst #AFB_<option>,D0 after having ↩
    called
GetUserFlags(). Example:

  JSRGEN  GetUserFlags
  btst  #AFB_NTSC,D0
  bne .nontsc

  ; ntsc set
  ; do stuff for NTSC ...

.nontsc


See also: GetUserData()
```

1.18 SetLocalVarZone()

```
void SetLocalVarZone(start,end) - Sets saveable variable location in the loader
                       A0  A1
```

```
When you make a snapshot, you save JST internal memory and the game internal  ↩
    memory,
but you may also need to save some data relevant to your loader (e.g. current disk ↩
    ).

You may not need this routine if your loader is completely state-independent
(i.e. does not require variables to work)

Extension memory pointer does not count as it's restored by the snapshot.
No need to save this one.

Always call this routine before SAVE_OSDATA (or use the SET_VARZONE macro)

See also: InGameIconify, SET_VARZONE (macro)
```

1.19 SetFilesPath()

```
void SetFilesPath(dirname) - Changes default path for files to load to wished  ↩
    value
                    A0

The first argument of HD_PARAMS can no longer be the subdir where the data files  ↩
    are
to be loaded.

Use SetFilesPath instead. This function will change the load directory.

See also:
```

1.20 UseHarryOSEmu()

```
void UseHarryOSEmu(void) - allows the use of Harry's excellent OS emulation  ↩
    program

This function will allow you to use Harry's OS emulation program.
This program must be called OSEmu.400 and be located in the C: directory.

This function allows a very powerful emulation, actually much better and complete
than my OpenFakeExec() function!!

Read the documentation in the source code of OSEmu.asm for more details and refer
to the JST loaders examples.

QUITKEY tooltype allows to change the default quit key (which is '*' of the  ↩
    numerical
keyboard, provided that the keyboard is US/German. Actually this key is the upper  ↩
    right
key of the numeric keypad, sorry A600 users :) )

See also: OpenFakeExec()
```

1.21 DisableChipmemGap()

```
void DisableChipmemGap(void) - deactivates gap in memory shadowing

When you call the macro SAVEOS_DATA with a maxchip value as argument,
let's say for instance SAVEOS_DATA $80000, JST will allocate $80000 bytes
of fastmem or chipmem (address above $80000 at all rates) to save the
chipmem between $0 and $80000, and to swap it when the loader quits or
swaps between game and OS (read/write data from HD...)

If the maxchip value is above $1FE000, JST does not normally allocate 2
megabytes of fast memory because it's useless to waste that much memory,
and uses AllocMem() on chipmem to allocate the largest chipmem block possible.
This block being allocated, there's no need to save all the 2MB of memory but
just below the lower bound of the allocated block and above the upper bound
of the allocated block.
But there's a technical problem here: when the loader reads/writes from the HD,
it's rather hard to know where the data will be written/read from, because
there's a gap which is not "shadowed", so I made this routine which forces JST
to allocate 2 megabytes to save the chipmem, without the gap trick.

It costs more memory, but allows proper OS swap and data read/write.

This routine has no effect (at the moment) if the argument passed to SAVEOS_DATA
is below $1FE000.

See also: SaveOSData()
```

1.22 ReadUserDir()

```
ULONG ReadUserDir(dirname, buffer, maxentries, maxnamelen) - Reads dir from the ↩
   user dir
 D0                A0       A1       D0          D1

This function will read the files in the directory specified in A0. The SAVEDIR ↩
   value
will be concatenated in the path if SAVEDIR is provided in the tooltypes (OS 2.0+ ↩
   only)

The filenames will be copied in the output buffer specified in A1.
Please note that only the plain files will be copied. Directories will be ignored.

This function will return the number of plainfiles in the directory in D0.

maxentries will allow to limit the number of files to be copied. It will prevent a ↩
    buffer
overflow. Set it to 0 if you don't want any limit.

maxnamelen will allow to limit the filename length. Set it to 0 and the limit will ↩
    be 20
characters.
The filenames will be NULL terminated in the buffer, and spaced by maxnamelen ↩
   bytes, even
if the filenames are not that long, to allow easy search in the buffer.
```

```
e.g set maxnamelen to 32 ($20), and call ReadUserDir. The buffer will look like:

$00: f i r s t _ f i l e\0 *rubbish*
$20: s e c o n d _ f i l e\0 *rubbish*
$40: t h i r d _ f i l e\0 *rubbish*

and so on...

I found it more convenient than a buffer depending on the actual length of the  ↩
    filenames,
because a simple MULU allows to access any filename.


See also: InGameOSCall()
```

1.23 SaveOSData()

```
void SaveOSData(chipmem_size,memsave) - Saves chipmem and custom & cia registers ↩
    ...
                     D0        D1

Usually called by the SAVE_OSDATA macro.

The most important routine of JST package. You've got to understand it
fully if you want to make a clean loader.

Useful for debug purposes and to be able to quit the game and make in-game
OS calls (read/write files)

The memsave parameter is only used when the chipmem_size is $200000
(AGA game loaded). If set to 0, SaveOSData will alloc the bigger chipmem
chunk as possible and will therefore not care about this part of memory,
so the save buffer in fastmem will be a lot smaller than 2MB.
If set to -1, all chipmem will be saved. If DEBUG is set, memsave
will also be set to -1 and you'll need more memory.

To save 1024K of chipmem:

     SAVE_OSDATA     $100000

To save 2048K of chipmem, trying to reduce fast memory consumption:

     SAVE_OSDATA     $200000


If no memory is available, the chipmem won't be saved, and it won't be possible
to quit the game/to get debug information.
As a side effect, this function disables all interrupts/DMA this way:

    lea      #$DFF000
    move.w   #$4000,intena(A6)
    move.w   #$7FFF,intreq(A6)
    move.w   #$4020,dmacon(A6)   ; also remove sprite DMA
```

```
This is necessary as the system could keep on modifying and restoring chipmem  ↩
    could
crash the machine. If your game bootup does not like the interrupts to be disabled ↩
    ,
reactivate necessary interrupts/dma yourself.

SAVING MEMORY FOR A 2MB CHIP GAME

(If you can avoid to use the full 2MB of chip do it)

Trying to save 2MB of chip will activate the memsave option.
This option consists in allocating as much as chipmem as possible, to avoid
to allocate a whole block of 2MB fastmem. The allocated chipmem can be trashed
by the game, it will not corrupt the system.

Be careful when you read or write a file to hard disk, because data may not match
(addresses are translated to the fastmem chip mirror during an OS swap
and the gap caused by the memsave option can 'corrupt' data).
If you plan to activate memsave and read/save files to HD, do the disk IO
in a fastmem buffer rather than in chipmem, and transfer it later (that's
why you can read the HDLOAD/LOWMEM flag in the user code)

With diskfiles, check and display an error message if LOWMEM is on, because of
the same problem, but you can't be sure if the bigger load part the game can
do at a time...

See also: GO_SUPERVISOR, InGameExit, InGameOSCall, ReadFileHD, WriteFileHD
```

1.24 Reboot()

```
Just reboots the computer (the OS must be active)
```

1.25 AllocExtMem(), Alloc24BitMem()

```
ULONG AllocExtMem(bytesize) - Allocates memory for expansion memory
ULONG Alloc24BitMem(bytesize) - Allocates memory for expansion memory in 24 bit  ↩
    space
 D0                 D0

Lots of games need more than the standard 512K of chip memory.
They often do some tests to know where they can write (see the patch section
in the guide).
Sometimes it's better not to leave the game detect the expansion RAM itself.
Using AllocExtMem, you'll allocate a block of 512K, 1024K or any size, and
you'll keep it in mind until the game searches for memory. Patch the routine
with the pointer you allocated and:

1) You'll be sure the game uses fast memory
2) The quit option will be safe, which means the memory will not be corrupted
   by the game.

You could do this allocation by hand but as you always quit by calling InGameExit ↩
    (),
```

```
you could not get control again to free the memory.
AllocExtMem() stores the data useful for FreeMem, and InGameExit() tries to free
the memory you allocated automatically.

If you select the NOFAST tooltype, the extension memory block to be returned
will ALWAYS be $80000, and more chip memory will be saved when you call
SaveOSData (no need to add). E.g:

        move.l        #$80000,D0
        JSRABS        AllocExtMem
        RELOC_MOVEL   D0,ExtBase   ; relocatable macro for move.l

  ...

  SAVE_OSDATA $80000


WARNING: Do not call this routine more than once, as the first allocated zone won' ↩
   t
be freeable.

Alloc24BitMem is exactly the same function except that it will try to allocate  ↩
   memory
with the 24BITDMA flag set (chip of 24 bit fastmem if you've got some). Useful for
some games that don't like 32 bit fastmem (shitty coded).

Both will check that the beginning of the allocated block is above address $80000,
and will return 0 if it's below.

See also:
```

1.26 OpenFakeExec()

```
ULONG OpenFakeExec(void) - Create a exec-like function table
 D0

This routine (new since V0.7 of JST) allocates $300 bytes and creates a function
table that matches ExecBase offsets. Of course all calls are not emulated, and  ↩
   some
are forbidden. For instance, you cannot OpenLibrary() (of course), or OpenDevice
with device name different from "trackdisk.device".

To use only after SaveOSData. Fake ExecBase will be in $4.
For the moment it does not support ExecBase structure. Only functions are  ↩
   supported.

This function table saves a lot of hassle because you don't have to insert BRA.B
everywhere to skip those function calls.

Calls supported:

CacheClearU (flushes caches)
CacheClearE (flushes caches)
CacheControl (flushes caches, returns 0 in D0 and D1)
AllocAbs (returns the value in A1 in D0, as if absblock had been allocated)
```

```
DoIO (calls TrackLoad(), as only trackdisk.device can be opened. Returns 0)
OpenDevice (checks for "trackdisk.device", and then returns 0)
FindTask (returns 0 in D0!!)

DoIO() is limited to commands 2 and 9. Write is not supported. If you want it to
be, you have to overwrite the DoIO function with SetFakeFunction().

Inactivated calls:

Disable
Enable
Forbid
Permit
SuperState
UserState
AddPort
RemPort
CloseLibrary
CloseDevice

Calling any of the above will only do a RTS

Unsupported calls:

OpenLibrary
OldOpenLibrary
AllocMem
FreeMem
AvailMem

Calling unsupported calls and other calls will trigger an runtime error.
The difference is that calling an unsupported call will display a specific message
(AllocMem not defined...) whereas other calls are not differentiated in the error
message (sorry you have to hunt for the offset)

Anyway, you can overwrite the system calls with the SetFakeFunction() call.
This way, you can define your own AllocMem(), etc...

If you succeed in defining some useful and cool routines, I'll be glad to  ↩
   integrate
them in JST default fakeexec emulation.

Return Values:

D0: 0 in any cases at the moment, but soon will return 0 if OK.

Note:

If DEADLY tooltype is on (DEADLY sets $FF000001 in ExecBase location to detect  ↩
   games
which use system calls besides other things), OpenFakeExec will display a warning.
DEADLY ExecBase kill will be deactivated.

Sometimes the game just reads in ExecBase to see the first block of fastmem, does  ↩
   calculations
on it with ANDs #$FFF80000 and so on. In this case you have to patch the routine  ↩
   by hand.
```

```
Just run the game from floppy or without OpenFakeExec or DEADLY to understand how ↩
    it behaves,
and then copy the behaviour.
The best thing to do is to have a stock A500 with Action Replay. Then you can see ↩
    exactly how
the game understands and stores the info for 512K chip and 512K fast.


See Also: SetFakeFunction(),TrackLoad()
```

1.27 FreezeAll()

```
void FreezeAll() - Freeze DMA and interrupts

Be sure to save the custom registers before with SaveCustomRegs.
This function is useful to start/stop a game without being annoyed
by ongoing DMA on interrupts.

See also: SaveCustomRegs(),RestoreCustomRegs()
```

1.28 LoadDisks()

```
void LoadDisks() - Load Disks in memory

Loads the disk from the HD_PARAMS macro parameters in the user program
no parameters are returned. The routine will exit with an error message
if something fails (no memory, file not found...)
else, it will return to the caller.

See also: LoadFiles(), LoadDiskFromName()
```

1.29 LoadDisksIndex()

```
void LoadDisksIndex(index) - Load Disks in memory, starting from index
                      D0

Same thing as LoadDisks(), but it begins at the disk number specified in D0

See also: LoadDisks(), LoadDiskFromName()
```

1.30 LoadDiskFromName()

```
void LoadDiskFromName(file name) - Load the file as a disk file
      D0

Loads the disk from the HD_PARAMS macro parameters in the user program,
but the name will be ignored and replaced by the one you mention.
```

```
no parameters are returned. The routine will exit with an error message
if something fails (no memory, file not found...)
else, it will return to the caller.

If you call it twice, first file will be disk 0 and second will be disk 1.

See also: LoadDisks(), LoadFiles()
```

1.31 LoadSmallFiles()

```
void LoadSmallFiles(size_limit) - Load small files in the current directory
                      D0

This function will load all the files which length is inferior to
size_limit. The other ones will be loaded during game, but it is
transparent to the programmer because the ReadFile() function will
read either from preloaded files or from hard disk if the file is
too big to have been cached.
You'll understand that you'll have to adjust properly size_limit
depending on the game you're patching. If a game directory holds
30 files of 1000 bytes each, and some big files (>50Ko)
you should set size_limit over 1000 else the ReadFileHD will be
called very often and as OS swaps take time and blackout the display,
your loader won't use properly the HDLOAD feature.

NOTE: If HDLOAD is not set, this function will behave exactly as
LoadFiles(), and will read all the files (no size limit)

See also: LoadDisks(), LoadFiles(), ReadFile()
```

1.32 LoadFiles()

```
void LoadFiles() - Load all the files in the current directory


This function will perform a Lock() in the current directory, then will
create the structures to store the files and read them all in memory.
It does not load subdirectories. Only flat files will be read, so be
careful with games storing data in subdirectories. The installation
procedure will have to flatten the file structure.

This function will exit with an error message if an error occurs
else, it will return to the caller

See also: LoadSmallFiles(),LoadDisks(),ReadFileFast(),LoadRNCFile().
```

1.33 GetFileLength()

```
ULONG GetFileLength(filename) - Gets the length (bytes) of a file
  D0                   D0

This function returns the length of a file in bytes. If the file is not found,
it will return -1, so be careful to test the value.

This function can only be called when the OS is active (not during game,
or if you must do so, do it within a InGameOSCall).

See also: ReadFile()
```

1.34 TestFile()

```
ULONG TestFile(filename) - Tests if a file exists on the hard drive
  D0               D0


This function returns 0 if the specified filename exists (relative path
from JST path)

See also: TestFileAbs
```

1.35 TestFileAbs()

```
ULONG TestFileAbs(filename) - Tests if a file exists on the hard drive
  D0               D0


This function returns 0 if the specified filename exists (absolute path)

See also: TestFileAbs
```

1.36 LoadRNCFile()

```
ULONG LoadRNCFile(filename) - Loads and decrunches a RNC file from the Hard Drive.
 D0/A0/D1            D0

This function loads a file and decrunches it, allocating memory at the same time  ↩
    for the
decrunched length.
Return values: D0: error flag (0 if OK)
               A0: beginning of allocated buffer
               D1: allocated length (useful if the file space must be freed with  ↩
                  FreeMem())

See also - RNCDecrunch(), RNCLength()
```

1.37 TransfRoutines()

```
void TransfRoutines(void) - (obsolete)

This routine is obsolete and provided only for compatibility reasons.
It does strictly nothing.

See also:
```

1.38 BlockFastMem()

```
void BlockFastMem(sparemem) - Blocks fastmem
                      D0

D0 is the number of bytes to leave unallocated (roughly, not accurate).
0 in D0 means the routine will try to allocate all fast memory available.

This is useful for some games which want chipmem but forget to specify it
in AllocMem() (old games), and also for games which do not like memory
located over $FFFFFF (coded in 32 bits).

Do not use it if not necessary (impossible to return to the OS
after that). You can use it when the TUDE NOFASTMEMORY BOOT=HIGHCHIP
is necessary to boot the wanted game from floppy without problems
TUDE is copyright N.O.M.A.D 1995. It can be found on aminet.

Normally if you track the game memory allocation, you should not use this
routine. Force the game to allocate its extension memory block in chipmem
(provided you've got 1MB chip, see BodyBlows patch).

See also: Degrade(), DegradeCpu().
```

1.39 Degrade()

```
void Degrade(cacheset,cachemask) - Degrades everything to run games properly
                D0       D1

This routine is a combination of both DegradeCpu() and DegradeGfx() routines.
Call it just before saving OS data and booting the game.

A classic sequence is:

  moveq #0,D0
  move.l  #CACRF_CopyBack,D1
  JSRABS  Degrade     ; leave only instruction cache

  GO_SUPERVISOR
  SAVE_OSDATA $80000

  .. boot stuff
```

```
See also: BlockFastMem(), DegradeCpu(), DegradeGfx(), Enhance()
```

1.40 Enhance()

```
Resets system enhancements (VBR, caches, gfx...)

This function should *NEVER* be called when using the library, but it could be
useful to call it to see what happens in an OS swap that fails. It's a function
that can be used in the standalone degrade code I've written

See also: Degrade()
```

1.41 DegradeGfx()

```
void DegradeGfx() - Degrades graphics to run games properly

This routine degrades the display using LoadView(NULL), and can set PAL or NTSC ( ↩
    tooltype).
It also opens an intuition screen if possible.


See also: BlockFastMem(), DegradeCpu(), Degrade()
```

1.42 DegradeCpu()

```
DegradeCpu(cacheset,cachemask) - Degrades only cpu related hardware
                 D0       D1

This routine does the following:

- It sets the VBR to $0 for 68010 and higher cpus unless the LEAVEVBR tooltype is  ↩
    present.
- It removes the specific 68060 caches (branch, writeallocate, and also the data/ ↩
    copyback
  cache, this last one the CacheControl() function leaves alone, even if you want  ↩
      to disable
  CopyBack)
- If NOCACHES is activated in the tooltypes, it will disable ALL caches.

The cacheset and cachemask variables are the same as in exec CacheControl().
Refer to the CacheControl autodoc for more details.

This function does not affect graphics modes at all.

See also: Degrade(), DegradeGfx()
```

1.43 Display()

```
void Display(message pointer) - Displays a message in the opened console or CLI
               A1

This function is generally called from the Mac_printf macro. This is more  ↩
    convenient,
but we may need to call it directly in some special cases.
If the program has been run from CLI, Display() will write in the CLI. If run from
workbench, it will write in the window opened by the startup program (see  ↩
    HD_PARAMS macro).

If no window exists, this function will do nothing (it won't even crash).
```

1.44 CloseAllQuiet()

```
void CloseAllQuiet(void) - Frees everything and exits


This function frees all the memory allocated by the program (diskfiles, files,  ↩
    relocatable
code, windows) and quits the program. If you allocated memory from the user  ↩
    program
by using directly Exec routines, you'll have to free it before calling this  ↩
    function.
This function cannot be called when the game is running (the OS is killed). Call  ↩
    InGameExit
instead.
This function will exit at once, and the window will be closed if the program was  ↩
    started
from Workbench.

See also: InGameExit(), CloseAllQuiet()
```

1.45 CloseAll()

```
void CloseAll(void) - Frees everything and exits because of an error


This function frees all the memory allocated by the program (diskfiles, files,  ↩
    relocatable
code, windows) and quits the program. If you allocated memory from the user  ↩
    program, you'll
have to free it before calling this function.
This function cannot be called when the game is running (the OS is killed). Call  ↩
    InGameExit
instead.

If the program has been run from workbench, it will ask for the RETURN key before  ↩
    closing
the console, in order for you to see what went wrong.
```

```
See also: InGameExit(), CloseAllQuiet()
```

1.46 FlushCachesSys()

```
void FlushCachesSys(void) - Flushes the caches using OS calls

Well, this function calls CacheControl(0L,0L), and the caches are flushed.

See also: FlushCachesHard(),Degrade()
```

1.47 GetMemFlag()

```
ULONG GetMemFlag(void) - Return MEMF_REVERSE if kick 39+
 D1

As the MEMF_REVERSE flag (AllocMem, AllocVec) does not exist in KS 1.x, and was ↩
    broken
until V39, this function checks for the kick version, and returns 0 if it is too ↩
    old.
Else, it will return MEMF_REVERSE.

That will explain that JST loaders will have more chances to work on V39 machines.
```

1.48 TestxMbChip()

```
ULONG Test1MBChip() - Hardware test for 1MB chip memory
ULONG Test2MBChip() - Hardware test for 2MB chip memory

This routine Test1MBChip (resp Test2MBChip) pokes in the location $80000 (resp ↩
    $100000),
then checks for location $0. If they are the same, then chipmem is 512K
(resp 1024K or 512K) ,else, it's at least 1024K (resp 2048K)
This routine is safe. It calls Disable and restores the altered chip locations.
Returns 0 if chipmem is as required (>1024K resp 2048K), -1 else.

NOTE: To test for 512K chip, use Test1MBChip. To check for 1MB chip, use
Test1MBChip, then Test2MBChip to know if it's only 1024K or 2048K.
I don't think further amigas (if there any) will have more than 2MB of chipmem...
Also notice that some computers have 2MB of chipmem and are not AGA, and
that some so-called AGA games only assume you've got 2MB of chip and can
be run on 2MB chip ECS amigas or even on 512K OCS amigas (but in that last
case you'll have to relocate some code heavily!!)

Example: Jungle Strike AGA (Ocean) runs fine on UAE with 2MB of chipmem.


See also: CheckAGA()
```

1.49 CheckAGA()

```
ULONG CheckAGA(void) - Checks DeniseID to see if the computer supports AGA or not
  D0

This function will return 0.L in D0 if AGA is supported, -1.L else.
```

1.50 GetSR()

```
UWORD   GetSR() - Returns CPU status register
 D0

SR will be copied in D0. No need to be in suprevisor mode to call
this function (or else it has no interest).

See also:
```

1.51 GetAttnFlags()

```
UWORD   GetAttnFlags() - Returns CPU AttnFlags while OS is down
 D0

At startup, exec's AttnFlags is copied and relocated. So if you need
to know something about the CPU you patch is running on, use this
function. You can test the bits like a normal CPU bit test.

See also:
```

1.52 WriteFileHD()

```
                        WriteFileHD(command,length,name,buffer) - Writes a file to ↩
                           disk during game
D0,D1                D0(=0)   D1    A0    A1

This function is very useful to write some data from disk on demand during
the game for savegame data files or hiscores.

D0.L: command: leave to 0 for the moment
D1.L: length in bytes (be careful!!)
A0:   pointer on the name, NULL terminated
A1:   source data buffer (either in fastmem or chipmem)


Note: as the game chipmem is transfered to a fastmem area during OS calls,
if you load a file into a chipmem zone, it will be loaded in the fastmem
mirror instead, and will appear in chipmem after the OS is killed again,
so don't worry about that (except if you save 2MB chip and activate memsave).

The function returns D0=0 on success, -1 else.
```

```
If success, D1.L contains the length actually written.


You can call this function whether the OS is alive or not.


If you want to write user data (like scores or gamesaves), you'd better use
                WriteUserFileHD()
                because this function uses SAVEDIR parameter to write data to  ↩
                   another directory in the
registered version of JST.

See also: InGameOSCall(), WriteUserFileHD().
```

1.53 WriteUserFileHD()

```
       WriteUserFileHD(command,length,name,buffer) - Writes a file to disk during  ↩
          game in SAVEDIR
D0,D1                   D0(=0)   D1    A0    A1


Same thing as WriteFileHD, except that if SAVEDIR tooltype is set, (SAVEDIR-RAM:), ↩
    the data
will be saved here, and not in the game directory. Useful for write only/copyback  ↩
   cached
partitions.

See also: WriteFileHD(), ReadUserFileHD().
```

1.54 ReadUserFileHD()

```
       ReadUserFileHD(command,length,name,buffer) - Reads a file to disk during  ↩
          game in SAVEDIR
D0,D1                   D0(=0)   D1    A0    A1


Same thing as ReadFileHD, except that if SAVEDIR tooltype is set, (SAVEDIR-RAM:),  ↩
   the data
will be saved here, and not in the game directory.

See also: ReadFileHD(), WriteUserFileHD().
```

1.55 ReadFile()

```
                      ReadFile(command,length,name,buffer) - Reads a file from  ↩
                         disk/RAM during game
D0,D1           D0(=0)   D1    A0    A1
```

```
This command is very similar to
                ReadFileHD()
                 except that it will
check if the file was not cached in memory first.

In the case HDLOAD/LOWMEM is on, it will often read files from HD. If those flags ↩
   are off,
all the files will be loaded from memory because they had been preloaded
(by LoadFiles() or LoadSmallFiles()).

See also: ReadFileHD(),ReadFilePart().
```

1.56 ReadFilePart()

```
                      ReadFilePart(command,length,offset,name,buffer) - Reads a ↩
                         part of a file from disk/RAM during game
D0,D1              D0(=0)   D1     D2    A0    A1

This command is very similar to
                ReadFilePartHD()
                 except that it will
check if the file was not cached in memory first.

In the case HDLOAD/LOWMEM is on, it will often read files from HD. If those flags ↩
   are off,
all the files will be loaded from memory because they had been preloaded
(by LoadFiles() or LoadSmallFiles()).

See also: ReadFileHD(),ReadFilePartHD().
```

1.57 ReadFilePartHD()

```
      ReadFilePartHD(command,length,offset,name,buffer) - Reads a part of a file ↩
         from disk/RAM during game
D0,D1                 D0(=0)   D1     D2    A0    A1

This function will try to load a part of a file from HD.

D0: 0 if OK
D1: length read in bytes

See also: ReadFileHD(),ReadFilePart().
```

1.58 ReadFileHD()

```
                      ReadFileHD(command,length,name,buffer) - Reads a file from ↩
                         disk during game
D0,D1            D0       D1    A0    A1
```

```
This function is very useful to read some data from disk on demand during
the game. It can load the game files or savegame data files.

D0: command: 0: normal operation, $FF reserved. Leave to zero at the moment
D1: length in bytes. -1 means all the file
A0: pointer on the name, NULL terminated
A1: destination data buffer (either in fastmem or chipmem)

Note: as the game chipmem is transfered to a fastmem area during OS calls,
if you load a file into a chipmem zone, it will be loaded in the fastmem
mirror instead, and will appear in chipmem after the OS is killed again,
so don't worry about that (except if you save 2MB chip, in that case,
I allocate chipmem to save some fastmem, so there's a gap!!)

The function returns D0=0 on success, -1 else.
If success, D1.L contains the length actually read.

You can call this function whether the OS is alive or not.

This function will not check for cached file in memory. Calling ReadFileHD forces
a read from HD (an OS swap is necessary). If you don't know if the file was cached ↩
    or
not, use
                ReadFile()
                 instead.

If you want to read user data (like scores or gamesaves), you'd better use
                ReadUserFileHD()
                because this function uses SAVEDIR parameter to read data from  ↩
                   another directory in the
registered version of JST.

See also: ReadFile(), InGameOSCall(), WriteFileHD(), DeleteFileHD(),  ↩
   ReadUserFileHD().
```

1.59 DeleteFileHD()

```
                ULONG DeleteFileHD(name) - Deletes a file from disk during game
 D0                 A0

This can also be done by calling InGameOSCall() but I needed it in the
Sensible Golf patch and then I included this function in the library.

Will return 0 if success, -1 else.

You can call this function whether the OS is alive or not.

If you want to delete user data (like scores or gamesaves), you'd better use
                DeleteUserFileHD()
                because this function uses SAVEDIR parameter to delete data from  ↩
                   another directory in the
registered version of JST.
```

```
See also: InGameOSCall(), ReadFileHD(), WriteFileHD(), DeleteUserFileHD()
```

1.60 DeleteUserFileHD()

```
ULONG DeleteUserFileHD(name) - Deletes a file from disk during game in SAVEDIR
 D0                     A0

Same routine as DeleteFileHD(), but uses SAVEDIR if set to search file to delete
into. Unvaluable because cannot be reproduced using InGameOSCall() (because
for the moment it's not possible to get SAVEDIR value from the object)


See also: InGameOSCall(), ReadFileHD(), WriteFileHD(), DeleteUserFileHD()
```

1.61 InGameOSCall()

```
ULONG InGameOSCall(entrypoint) - Calls a routine containing OS code
                       A4

This is one of the most powerful and also one of the most dangerous function of
this package.
It allows the user to call a program using normal system calls, while the game
has totally destroyed the OS.
It restores the OS and interrupts and saves the game memory, goes in user mode,  ↩
    calls
the user routine, and restores everything for the game.
The display goes black because this is not possible to save the colormap and other
screen data.
The user routine can trash all the registers. They're saved and restored after the ↩
     call.

However, there are problems with this routine:

- Any Write() made to disk may not terminate when the game is re-activated. So
  you'll better disable disk caches and quit the game correctly after that.
  If someone knows how to wait for the disk activity to finish, please call me.
  The problem that may occur is a devalidation of the disk (not very important).
  I use the dos Delay() command after the write operation with 2 seconds, and it  ↩
      works:

  move.l  #100,D1
  move.l  _DosBase(pc),A6
  JSRLIB  Delay

- Playing with the keyboard while the disk is writing or reading locks the system.
  I don't know where it comes from, but I'll find out. For the moment, avoid to  ↩
      press
  any key during disk accesses (especially writes, this can lead to write errors  ↩
      !!!)

Besides those problems this routine is the only way to make real hard disk  ↩
    versions of games
```

```
which load and save data to disk.
Lots of the patches I've written use this routine. It took me some time for it to ↩
   be reliable.
Some users still have problems. I'm improving it from time to time.


This routine returns 0 when the routine has been called, and -1 if the routine has ↩
    not been
called because of the impossibility to save/restore the operating system, so be ↩
   careful for
users without too much memory...

The register D0 you'll set in the OS routine will be passed to your program. It ↩
   must be coherent
with the error code. E.g: if everything is OK, return 0, else return -1, or -2 to ↩
   make a difference
with the OS switching error.

Example:

LoadScoreFile:
  lea scorename(pc),A0
  move.l  A0,D1
  move.l  #MODE_OLDFILE,D2
  move.l  _SysBase(pc),A6
  JSRLIB  Open
  tst.l D0
  beq error$

  ... ; do our stuff (reading...)

  JSRLIB  Close
  moveq.l #0,D0
  rts
error$
  moveq.l #-2,D0  ; file error code
  rts

LoadHighscores:
  lea LoadScoreFile(pc),A4
  JSRGEN  InGameOSCall
  tst.l D0
  beq ok$   ; all is OK
  cmp.l #-1,D0
  beq oserr$    ; os could not be restored
        ; else file error

I recently added the ReadFileHD() and WriteFileHD() routines, and this
routine is not to be used in most cases, but sometimes it's compulsory
(for instance listing a savegame directory during the game (Cannon Fodder 2))


IMPORTANT: This function will do nothing if the NOOSSWAP tooltype/argument is ↩
   activated. To check
this argument, the register D1 will be set (-1.L) when _loader is called ( ↩
   beginning of the user
code).
```

See also: ReadFileHD(), WriteFileHD(), DeleteFileHD(), InGameExit()

1.62 PatchExceptions()

void PatchExceptions(void) - redirect system exceptions to custom routines

This routine allows to trap the following processor exceptions:

- Bus error
- Address error
- Illegal instruction
- Division by zero
- LINEA emulation
- LINEF emulation
- Format error
- some others... (subject to change)

PatchExceptions() is called at startup, so there's no need to do it by hand
normally, but it may be necessary to call it again in the user program in the
case of the game overwriting the values, and then crashes. That's why this ↩
function
is callable from user program.

When an exception is encountered, the handler returns to the OS with the ' ↩
exception
occured' or 'linef/movep (060)' message. If you want more information, use the ↩
DEBUG
tooltype or argument when running the loader, and a memory image of the game,
the registers an some other information will be written to disk. With the memory
and the stack value, maybe you're able to track the error or find out problems ↩
using
memory dump.

See Also:

1.63 FlushCachesHard()

void FlushCachesHard(void) - Flushes the caches

This routine is *VERY* useful to flush the caches when the OS is overridden
by the game, and you still want to use fast memory with the caches on.
It addresses CACR register to invalidate instruction and data caches,
and it also calls CPUSH to flush the copyback data cache into memory
(for 68040/68060) caches.

NOTE: As long as the OS is alive, use FlushCachesSys() whenever possible.

See also: FlushCachesSys()

1.64 GoECS()

```
void GoECS(void) - degrades display and sprites to ECS

This function is very useful when you start the loader from a multiscan screen ↩
    with
a hires pointer. In that case, if the game uses sprites, they all appear shrinked,
as the game did not switch in lores pointer.
It modifies the value of FMODE (set to 0) and BPLCON3 (set to $0C20).
This acts also on dual playfield parameters.

NOTE: This function will have no effect if the system copperlists are still active ↩
    ,
as they reset BPLCON3 every screen scan. Use it in an early stage of the game  ↩
    loader.
Look at the examples provided.

GoECS() is just a call to ResetDisplay() followed by a call to ResetSprites().

See also: ResetDisplay(), ResetSprites()
```

1.65 ResetDisplay()

```
void ResetDisplay(void) - Resets 15KHz display

This function avoids to get a modulo/AGA display problem.

NOTE: This function will have no effect if the system copperlists are still active ↩
    ,
as they reset BPLCON3 every screen scan. Use it in an early stage of the game  ↩
    loader.
Look at the examples provided.

See also: GoECS(), ResetSprites()
```

1.66 ResetSprites()

```
void ResetSprites(void) - Resets LORES sprites


If the sprites are too small vertically, call this function.

NOTE: This function will have no effect if the system copperlists are still active ↩
    ,
as they reset BPLCON3 every screen scan. Use it in an early stage of the game  ↩
    loader.
Look at the examples provided.

See also: GoECS(), ResetDisplay()
```

1.67 KickVerTest()

```
ULONG KickVerTest(kickversion) - Test if ROM version is newer than a given version
 D0                    D0

KickVerTest() uses the SoftVer variable in the ExecBase structure to test the
kickstart version.

kickversion is the kickstart version number (for instance 35,36,39...)
The function will return 0 is the installed version is newer than the wanted  ↩
    version
and -1 if not.

See also: Kick37Test()
```

1.68 Kick37Test()

```
ULONG Kick37Test(void) - Test if ROM is V37 or newer
 D0

The same thing as KickVerTest(), but compares the version to V37 (matches  ↩
    Workbench 2.0).
Useful when you don't know if you can use 2.0 features.

Returns D0=0  if KS>36, D0!=0 if KS<=36

See also: KickVerTest()
```

1.69 InitTrackDisk()

```
void InitTrackDisk(void) - Initialize trackdisk.device emulation

Call this when you need to emulate a DoIO(), very frequent in the bootblock of
games, which use trackdisk.device to load their loading routines, then forget
it along with the remainder of the system.
When you re-source the bootblock, replace all the JSRLIB DoIO by JSRGEN  ↩
    TrackLoadFast.
Before jumping to the bootblock code, you MUST call InitTrackDisk (use JSRGEN too) ↩
    ,
or else the loads will be done anywhere in the memory -> GURU.
Refer to the examples for more details.

See also: TrackLoadFast(),SetTDUnit(),ReadRobSectorsFast()
```

1.70 TrackLoad

```
void TrackLoad(void) - Replaces DoIO function to read from a virtual floppy

This function replaces TrackLoadFast. It has the same function, but is able to
```

```
read from HD when LOWMEM is activated.

This function replaces the DoIO() function in the case of a floppy disk.
Once initialized with InitTrackDisk() and possibly with SetTDUnit(), you are
able to read sectors from the virtual floppy device exactly with the trackdisk. ↩
   device
(only read is supported. There is no write mode).

See also: InitTrackDisk(),SetTDUnit(),ReadRobSectors()
```

1.71 SetTDUnit()

```
void SetTDUnit(unit) - Sets trackdisk.device emulation unit
               D0

This function allows to change the current virtual drive (actually it changes  ↩
   disks).
Use it in conjunction with TrackLoadFast(). It has no effect with  ↩
   ReadRobSectorsFast().

The unit is not limited to 3, as you can patch games which have got 4 floppies or  ↩
   more.

If this routine is not called, the unit will be 0 (default, useful for bootblocks) ↩
   .

See also: InitTrackDisk(),TrackLoad()
```

1.72 ReadFileFast()

```
ULONG ReadFileFast(filename,buffer,command,length) - Transfer a file in the given  ↩
   buffer
 D0                  A0      A1      D0      D1

Uses the Rob Northern file interface.

used in:

Cannon Fodder 2, Darkmere, Sensible Golf, Sensible Soccer, Road Rash...

Different versions may exist: D1 can be meaningless. In this case, set it to -1.
Some games only use commands 0 and 5

Darkmere uses 0, 5, 6, 7, 8 extensively.

in:
A0: Filename
A1: Buffer
D0: command:
D1: misc

D0:
```

```
0: Read
  in : D1: # bytes (-1: all), A0: filename, A1: buffer
  out: D0: 0 if OK,  D1: length read

1: Write (disabled)
  in: D1: # bytes          , A0: filename, A1: buffer
  out: D0: 0 if OK,  D1: length written

2: Never seen it before ????
3: Read directory (disabled)                          A0:
4: Format floppy (disabled)
5: Get Length
  in : A0: filename
  out: D0: 1 if found, 0 else, D1: length

6: Nothing (I think)

7: Read last file w/ offset
  in : D1: # bytes          , A0: scratch  ,A1: buffer
    out: D0: ???

8: Set offset on last accessed file
    in : D1: offset           , A0: scratch  ,A1: scratch
    out: D0: offset

In order to use this function, you must initialize the fastfile structure by  ↩
   calling
LoadFiles(), else this will fail.

filename: a NULL-terminated string
buffer: a pointer on a memory zone
length: the length in bytes. If you pass -1, the whole file will be read.

Sometimes, the games use this routine as-is, and you've got only to patch.
However, you've got to check for files which are not found: they can be savegames  ↩
   loads.

In that case, you can call the original routine (read from floppy) or make a HD  ↩
   access
using InGameOSCall() (much harder, provided you've got to read the directory)


To read part of files (using the 7 and 8 commands), it's better to use  ↩
   ReadFilePart()
 because the interface is more natural. However, this routine can be used as-is in ↩
     some
games (such as Darkmere, which uses the 7 and 8 commands, and took me a while to  ↩
   figure
it out!)


See also: LoadFiles(),ReadFile(),ReadFilePart()
```

1.73 ReadRobSectors()

```
ULONG ReadRobSectors(drivenum,offset,blocks,command,buffer) - Reads sectors from ↩
   virtual disk
 D0                    D0      D1     D2     D3      A0


This is the main all-purpose loading routine of the package. I chose this syntax
because this routine is used as-is in lots of games and can be used provided
an offset correction or a disk choice is done in many games.
I noticed it and replaced it in:

Assassin, BodyBlows, AlienBreed, Warzone, Magic Pockets, Cadaver,
Chaos Engine, Cannon Fodder 2, Project-X, Mortal Kombat I & II,
Gods, Rodland, Desert Strike, Qwak, and others...

INTERFACE:

drivenum/D0 = Drive Number - sometimes matches disk number
offset/D1   = Offset in bytes / 512 (also offset in sectors)
block/D2    = nb of 512 bytes sectors to read
command/D3  = 0: Read data, !=0 Writes (has to be done manually, as it's better to ↩
    create a separate file for game saves)
buffer/A0   = Start address

For 2 disked games using DF0: and DF1:, you generally don't have to worry about ↩
   disk changes,
as disk1 is supposed to be in DF0: and disk2 in DF1:, so D0 will be set in a ↩
   correct way and
the game will read from the correct disk anyway.

See the examples to understand fully the interest of this routine.

Specifying the LOWMEM tooltype/argument at startup causes this
routine to read the data from the file instead of the cached images
in memory (which will not exist if LOWMEM is activated).

See also: TrackLoadFast(), LoadDisks(), ReadDiskPart()
```

1.74 ReadDiskPart()

```
ULONG ReadDiskPart(drivenum,length,offset,buffer) - Reads bytes from virtual disk
 D0                    D0      D1     D2     A0


Even if the Rob Northen format is widespread, this routine is the alternative.
It is able to read any number of bytes and at any location on a disk image. It's ↩
   useful
for disk loaders which use a non $200 round interface.


INTERFACE:

drivenum/D0 = Drive Number - sometimes matches disk number
```

```
offset/D1   = Length in bytes to read
block/D2    = Offset in bytes from where to read
buffer/A0   = Start address

Specifying the LOWMEM tooltype/argument at startup causes this
routine to read the data from the file instead of the cached images
in memory (which will not exist if LOWMEM is activated).

See also: TrackLoadFast(), LoadDisks(), ReadRobSectors()
```

1.75 StrcpyAsm()

```
void StrcpyAsm(string1,string2) - Copies string
                D0      D1

Copies the string pointed by D0 to the buffer area in D1.
The source string must be NULL terminated.

See also: StrcmpAsm(), StrlenAsm(), ToUpper()
```

1.76 StrcmpAsm()

```
LONG StrcmpAsm(string1,string2) - Compares 2 strings (not case sensitive)
 D0             D0      D1

StrcmpAsm returns 0 if the 2 strings are the same (no case sensitivity,
e.g. LoA-Der is the same as LOa-deR), and -1 if they're different

See also: StrcmpAsm(), StrcpyAsm(), ToUpper()
```

1.77 StrlenAsm()

```
ULONG StrlenAsm(string) - Returns the length of a NULL-terminated string
 D0              D0
```

1.78 ToUpperAsm()

```
void ToUpperAsm(string) - Converts a string in upper case
 D0              D0
```

1.79 HexReplaceLong()

```
void HexReplaceLong(searched,replacer,start,end) - Searches a longword and  ↩
   replaces it
                       D0       D1     A0   A1

Designed for automatic search/replace of data/code.

This function can be useful to search blitter commands (move.w Dx,($58,Ax)) to  ↩
   patch
them with WaitBlit. For example, I'd like to insert blitterwaits in the zone
$1000-$5000 for the instruction  move.w  D1,($58,A2) (hex: $35410058)

    ...
    move.l   #$35410058,D0
    move.l   #$4EB800C6,D1   ; JSR ($C6).W
    lea      $1000.W,A0
    lea      $5000.W,A1
    JSRGEN   HexReplaceLong
    PATCHUSRJMP    $C6.W,DoBlit_D1A2
    ...

DoBlit_D1A2:
    JSRGEN   WaitBlit        ; wait blitter operations to complete
    move.w   D1,($58,A2)     ; start the blit
    rts

The search will be done word by word (2 bytes). Odd occurences will not be found.

See also: HexReplaceWord
```

1.80 HexReplaceWord()

```
void HexReplaceWord(searched,replacer,start,end) - Searches a word and replaces it
                      D0.W    D1.W     A0    A1

Designed for automatic search/replace of data/code.

Same use as HexReplaceLong().

The search will be done word by word (2 bytes). Odd occurences will not be found.

See also: HexReplaceLong()
```

1.81 SetFakeFunction()

```
void SetFakeFunction(offset, new function) - Sets user defined function in  ↩
   fakeexec table
                       D0       A0

The exec library emulation is poor. You'll have to define your own functions to  ↩
   make
it better (game specific or not).
```

```
For instance, if a game uses AllocMem(), it's up
to you to choose what to return, to keep track of the allocated blocks or not
(is it worth? is AllocMem() called more than once, twice...?)

You could do it in a "dirty" way, by getting FakeExecBase in $4 and patching the  ↩
   offset,
but I created a function for this. As a bonus, the function will check:

1. That you pass an offset which is between 0 and -$300
2. That OpenFakeExec() was called (before SaveOSData)

If both conditions are not passed, the function will trigger an explicit run-time  ↩
   error.

Example: define your own AvailMem()

       ...

       lea    MyOwn_AvailMem(pc),A0
       GETLVO AvailMem
       JSRGEN SetFakeFunction
       ...

MyOwn_AvailMem:
       move.l   #$70000,D0
       rts

I used my new macro GETLVO to get an LVO in D0. (move.l  #_LVO\1,D0)

See also: OpenFakeExec()
```

1.82 CRC16()

```
UWORD CRC16 (buffer, length) - Calculates CRC16 checksum
 D0             A0       D0

This routine calculates CRC16 checksum of a block according to the
ANSI CRC16 algorithm.

Thanks to Andreas Kleiner for putting the crc.c source on aminet,
which I converted into assembler (available on request)

This routine was added because of the WHDLoad slave emulation.
```

1.83 HexToString()

```
void HexToString(number,buffer) - Converts a hex number to a ascii string
                   D0    A1

The buffer has to be at least as wide as 10 chars. The format will always be:
```

```
$xxxxxxxx.

e.g: D0=$37 gives A1-> $00000037
     D0=$DEADBEEF gives A1-> $DEADBEEF
```

1.84 Save & Restore hardware registers

```
void SaveCustomRegs() - Save readable custom registers
void RestoreCustomRegs() - Restore previously saved custom registers
void SaveCIARegs() - Save CIA registers
void RestoreCIARegs() - Restore CIA registers

Those functions are used internally to save/restore intena, intreq, adkcon
dmacon, and the CIA registers from/to the operating system.
So the game can (and will) trash those registers, but they will be restored
when InGameExit() is called.
You should not use those functions, unless you know what you are doing.

See also: InGameExit(), SaveOSData(), SAVE_OSDATA (macro)
```

1.85 RNCLength()

```
ULONG RNCLength(memory) - returns the length of a decrunched RNC file
 D0              A0

This function returns 0 if the zone pointed by A0 is not a RNC crunched file.
Else, it will return the number of bytes the file will take once decrunched.

See also - LoadRNCFile(), RNCDecrunch()
```

1.86 RNCDecrunch()

```
ULONG RNCDecrunch(crunchbuffer,decrunchbuffer) - Decrunches a RNC type 1 file
 D0                   A0            A1

This routine handles Rob Northen Cruncher files (header: RNC\01).
This type of cruncher is very heavily used in lots of games
from EOA, Team 17, Renegade, Akklaim, and lots of others.

A0 points on the crunched buffer start (the file just loaded), and
A1 points on the destination.

This routine returns 0 if an error occured.

The two addresses may be the same, as the decrunch algorithm is able to
overwrite the crunched data during decrunch.
Be careful to allocate or reserve enough memory to decrunch the file.
Use the RNCLength() function to know the length of the file once decrunched.
```

```
LoadRNCFile uses this routine, but RNCDecrunch can be useful when the game is  ↩
   NONDOS
and the data is trackloaded.

NOTE: RNC crunched files are handled by the XFD library (decrunch only)
The cruncher (ProPack) is not freely distributable, and reserved to
game programmers.

See also - LoadRNCFile(), RNCLength(), RNCDecrunchEncrypted(), ATNDecrunch()
```

1.87 RNCDecrunchEncrypted()

```
ULONG RNCDecrunchEncrypted(key,crunchbuffer,decrunchbuffer) - Decrunches a RNC  ↩
   type 1 file, with encryption
 D0                          D0      A0            A1

Some games, like Walker, use RNC compression, but encrypt data with a 16/32
bit key. You'll need to know the key to be able to decrunch the file
properly. In most cases, leave the game handle that. I use this routine
because in most games the decrunch routines are located in chipmem and it's
slow. You can patch the entry of a RNC decrunch routine by this call,
which is in fastmem, and you'll be amazed by the speed.

See also - LoadRNCFile(), RNCLength(), RNCDecrunch(), ATNDecrunch(), PPDecrunch(),
           FungusDecrunch()
```

1.88 ATNDecrunch()

```
ULONG ATNDecrunch(crunchbuffer,decrunchbuffer) - Decrunches a ATN! file
 D0                   A0            A1

This routine handles ATN files (header: ATN!).
This type of cruncher is very heavily used in lots of games
from Team 17 (Arcade Pool, Project-X...)

A0 points on the crunched buffer start (the file just loaded), and
A1 points on the destination.

This routine returns 0 if an error occured.

The two addresses may be the same, as the decrunch algorithm is able to
overwrite the crunched data during decrunch.
Be careful to allocate or reserve enough memory to decrunch the file.
Use the ATNLength() function to know the length of the file once decrunched.


NOTE: ATN crunched files are handled by the XFD library (decrunch only)
The cruncher is not available. This cruncher no longer appears in
Team 17 games and has been replaced by RNC.

See also - LoadRNCFile(), RNCLength(), RNCDecrunchEncrypted()
```

1.89 PPDecrunch()

```
PPDecrunch(end_crunchbuf,start_crunchbuf,decrunchbuf) - Decrunches a PP20 file
              A0              A1             A2

As I saw this routine in at least one game and I know PowerPacker
by Nico François is very popular on the amiga, I included this decrunch routine.

This applies to file beginning by the PP20 header (PowerPacker 2.0)

in: A0 end of source buffer
    A1 start of dest buffer
    A2 start of source buffer

out: nothing, but the buffer is decrunched :-)

All registers are preserved in this call.

See also - RNCDecrunch(), ATNDecrunch(), FungusDecrunch()
```

1.90 FungusDecrunch()

```
void FungusDecrunch(crunchbuf,decrunchbuf) - Decrunches a FUNGUS file
                         A0          A1

The FUNGUS packer (maybe also called SF or SA packer) is mostly used
in Gremlins games (Zool, Switchblade 2...)

in: A0: crunched buffer start
in: A1: destination (may be the same, like in RNC decruncher)

CAUTION: This decrunch routine does not check that the data is correct.
Unreliable results may occur if you try to decrunch a random block of memory.

See also - RNCDecrunch(), ATNDecrunch(), PPDecrunch()
```

1.91 BlackScreen()

```
void BlackScreen() - Sets all the colors to black

This function will not build a copperlist. It will just clear color registers.
```

1.92 EnterDebugger()

```
void EnterDebugger(void) - Enters debugger if one is installer

ATM only HRTMon is property supported. A version of HRTMon which works is for ↩
   instance
```

```
2.22. All versions above this one work too.

EnterDebugger calls the debugger just as if you put a breakpoint there, except  ↩
    that
you don't have too :)
VERY useful function when the loader is in development phase, to remove afterwards ↩
    !

This function will do nothing if no debugger was found. Run JST with VERBOSE or  ↩
    TEST on
to see which debugger is currently loaded.

See also:
```

1.93 SetTraceVector()

```
APTR SetTraceVector(APTR trace_entrypoint) - sets trace vector to a user routine
 A0                 A0

Now that JST relocates the VBR (unless you use the VBR-specific tooltypes), it's  ↩
    no
longer possible to poke directly in the trace exception vector to install a tracer
by poking in $24, since JST will intercept the trace exception with the relocated  ↩
    VBR.

This function allows to do it transparently.

in: A0 points to your trace code. Of course it's up to you to preserve registers  ↩
    on exit.
    Exit by RTE (not RTS)

out: A0 points to the old trace code. Most of the time it's the routine JST  ↩
    installed.

See also:
```

1.94 WaitMouse()

```
void WaitMouse(void) - Waits for LMB while the screen is full of colors

Useful to see if a point is reached. Waits until the LMB is pressed to exit.


See also: WaitMouseInterrupt()
```

1.95 WaitMouseInterrupt()

```
void WaitMouseInterrupt(void) - Waits for LMB while the screen is full of colors,  ↩
    interrupts active
```

```
Same routine as WaitMouse() except that as a bonus, interrupts will be enabled  ↩
   during the
wait, allowing you to break with a software debugger like HRTMon.

But this function can lead to crashes in some cases (e.g. the interrupts were  ↩
   disabled
because the interrupts vectors were not set by the game yet)

See also: WaitMouse()
```

1.96 InGameExit()

```
void InGameExit() - returns to the OS while the game is running

This function will return with a rts if no memory was available
for the chipmem (SaveOSData()). The best way to call it is from
a JSRGEN, then return to the program if it could not quit.

This function was moved from the absolute part to the relocatable part
in order to avoid crashes when the absolute program has be overridden
by the game. JSRABS InGameExit will not work (it will not crash in
most cases but will do nothing).

Now you can call this function before having called SaveOSData()
or the SAVE_OSDATA macro from the user program.

Call it whenever you want during the game, even if the OS
is totally killed. This function will resurrect the OS and will
call the user routine you specified using the SetExitRoutine function
if any. (see macro HDPARAMS)

This function has be greatly improved. It succeeds on 99.9% of games.

See also: SaveOSData(), InGameOSCall()
```

1.97 IsRegistered()

```
ULONG IsRegistered(void) - Check if the current version of JST is registered
  D0

Returns 0 in D0 if the version of JST currently used is not registered
Returns 1 in D0 if the version of JST currently used is registered (from v1.1e)

See also:
```

1.98 SetExitRoutine()

```
void SetExitRoutine(routine entrypoint) - calls a user routine on exit
                           A0
```

```
This function will allow the user to specify a function to call
when the OS has been restored with InGameExit(). For instance, this function
can save scores to a file without the use of WriteFileHD but in a normal DOS
way, or can display a message, free some manually allocated memory...

If you want to save scores on exit which are located in chip memory,
remember that you'll have to copy the scores in fastmem before calling
InGameExit, or the buffer will be trashed by the OS chipmem.

The user routine must end by RTS.

CAUTION: The input was D0 and now it's A0 because it's more convenient for
relocatable code.

See also: InGameExit()
```

1.99 InGameIconify()

```
                void InGameIconify() - returns to the OS while the game is running ↩
                   , with possibility of return

This function acts exactly like InGameExit BUT it allows to return to the game!
That seems impossible to do because of the read only registers, that's why some  ↩
   special
functions are to be applied on the game before you can use this function.

Actually, JST is already able to swap the OS but the display, because the copper  ↩
   pointer
is ready only.
But you can use some other functions to patch the game at some strategic locations
when it stores the copperlist pointer into the copper pointer register (COP1LC,  ↩
   $80).

Those functions modify the MOVE instruction which are used to store the
copperlist pointer very easily.

Copperlist pointer store can be of these types:

1:  move.l  Ax,($80,Ay)
2:  move.l  #chipaddr,($80,Ay)
3:  move.l  addr,($80,Ay)
4:  move.l  #chipaddr,$DFF080
5:  move.l  addr,$DFF080
6:  move.l  Ax,$DFF080

There are many ways to store a copperlist pointer, but those are the most frequent ↩
    occurences.

--

CASE 1:

To search (and patch) case 1, you can use
                HexReplaceLong()
                 because this instruction is
```

```
4 bytes long and is totally determined once you found which registers the game is  ↩
    using
(with HRTMon just search with fi and the pattern A*,($80,A*) )

Replace the instruction by $4EB8addr where $addr is a zero page address where you  ↩
    can
put the storage code or put a PATCHUSRJMP to a user routine (I prefer that second  ↩
    solution)

Be careful of the range, and check the locations you patch are actually copper  ↩
    moves
(just ensure Ay contains $DFF000), or else the iconify could go bezerk
(but the game will still work)

In that use routine, you'll put the value in the copper pointer and register it to ↩
     JST
using the SetCopperPointer() function. You've done it.

This is the most difficult and manual case, but when you're used to it, that's not ↩
     a concern
anymore.

CASE 2:

You lucky guys I made a special search and patch routine for those frequent  ↩
    occurences.
It's called
                PatchMoveCList_Idx()
                . There are a couple of parameters
but it's very easy to use and totally transparent.
You've got to put it at the right place, that's all.


CASE 3:

Like in case 2, I also made a routine called
                PatchMoveCList_Abs()
                CASE 4:

Like in case 2, I also made a routine called
                PatchMoveCList_Ind()
                CASE 5:

No routine implemented in JST. Totally manual method. I may add a  ↩
    PatchMoveCList_xxx function
later but now I don't feel like doing it.

CASE 6:

Well, totally manual method, but easy. Note down the addresses, and then make a  ↩
    PATCHUSRJSR
to your user routine (the instruction is also 6 bytes long)
which will do the poke in the copperlist register and will log
the copper pointer value to JST using SetCopperPointer().

--
```

```
If you cannot find in copperlist store instruction belonging to those 6 cases ↩
    above,
you'll have to search further, but game coders usually don't hide this kind
of instruction too much (it can happen, though, when sometimes they try to fool
the Action Replay cartridges).

You've got another solution if you don't find the copperlist pointer: look into ↩
    the game
using Action Replay cartridge (or equivalent) or do a copper search with HTRMon.
If the copperlist location does not move, you can hardcode it in your program and ↩
    call
SetCopperPointer() with this address.


When you logged all the copperlist pointer store instructions, InGameIconify can ↩
    be called
exactly like InGameExit, using for instance the TAB key in the keyboard interrupt.

In some cases, you'll have to set some hardware registers 'by hand' after JST ↩
    resumes the game.
For instance, I had to put a move.w #$400F,fmode+$DFF000 in Street Racer, or else ↩
    the display
was trashed.
But in some games like Menace or Lotus, the iconification works perfectly.

InGameIconify won't do anything if the current copperlist pointer is set to 0.
You can set it to this value with StoreCopperPointer()

See the examples (Menace, Street Racer, Lotus Turbo Challenge) to figure out how ↩
    to do
this.

Warning: QUIET will prevent this function to work. It will do nothing in that case ↩
    . Just
because JST uses the console in the iconify menu.

See also: InGameExit(), StoreCopperPointer(), TellCopperPointer(), ↩
    PatchMoveCList_Idx(),
           PatchMoveCList_Abs(), HexReplaceLong()
```

1.100 PatchMoveCList_Abs()

```
void PatchMoveCList_Abs(start, end, trap_number) - patches copper list stores: ↩
    move.l #adr,$DFF080
                          A0     A1     D1

This function puts a TRAP when it finds a MOVE.L #ADR,$DFF080 instruction.
ADR is checked so no patch is done if ADR is not in chipmem.

If D1 is out of $0-$F range, this function will do nothing. You can usually choose ↩
     any of those
numbers (which match trap numbers location $80 ->$BC), but some games use some ↩
    traps, and
```

```
obviously you cannot use the same ones. You'd be very unlucky if the game used ALL ↩
    the traps.
In that case, you've got to patch manually.

A0: start address: MUST BE AN EVEN ADDRESS!
A1: end address
D1: trap number ($0-$F)

This function flushes the caches at the end of the operation.

See also: PatchMoveCList_Idx(), PatchMoveCList_Ind(), InGameIconify()
```

1.101 PatchMoveCList_Ind()

```
void PatchMoveCList_Ind(start, end, trap_number) - patches copper list stores:  ↩
   move.l adr,$DFF080
                          A0     A1     D1

This function puts a TRAP when it finds a MOVE.L ADR,$DFF080 instruction.
ADR is checked so no patch is done if ADR is not in chipmem.

If D1 is out of $0-$F range, this function will do nothing. You can usually choose ↩
    any of those
numbers (which match trap numbers location $80 ->$BC), but some games use some  ↩
   traps, and
obviously you cannot use the same ones. You'd be very unlucky if the game used ALL ↩
    the traps.
In that case, you've got to patch manually.

A0: start address: MUST BE AN EVEN ADDRESS!
A1: end address
D1: trap number ($0-$F)

This function flushes the caches at the end of the operation.

See also: PatchMoveCList_Idx(), PatchMoveCList_Abs(), InGameIconify()
```

1.102 PatchMoveCList_Idx()

```
void PatchMoveCList_Idx(start, end, register#, trap#) - patches copper list stores ↩
   : move.l #adr,($80,Ax)
                          A0     A1     D0       D1

This function puts a TRAP when it finds a MOVE.L #ADR,($80,Ax) instruction if the
register number you specified matches Ax. It stores the copper pointer so  ↩
   InGameIconify
knows where it is.

ADR is checked so no patch is done if ADR is not in chipmem.
```

```
This function will do nothing if D1 is out of $0-$F range or if D0 is out of 0-6  ↩
   range.
Most of the time, you can choose any of those numbers $0-$F for D1
(which match trap numbers location $80 ->$BC) but some games use some traps, and
obviously you cannot use the same ones. You'd be very unlucky if the game used ALL ↩
    the traps.
In that case, you've got to patch manually (hard luck!)

A0: start address: MUST BE AN EVEN ADDRESS!
A1: end address
D0: register number (0-6)
D1: trap number ($0-$F)

This function flushes the caches at the end of the operation.

See also: PatchMoveCList_Abs(), PatchMoveCList_Ind(), InGameIconify()
```

1.103 PatchMoveBlit_Idx()

```
                void PatchMoveBlit_Idx(start, end, register#, trap#) - patches  ↩
                   blit moves: move.w #adr,($58,Ax)
                        A0     A1     D0       D1

This function puts a TRAP when it finds a MOVE.W #ADR,($58,Ax) instruction if the
register number you specified matches Ax.
The trap performs the blit but calls
                WaitBlit()
                 before, so the
blitter is always ready and there are no blitter errors due to those calls.

This function will do nothing if D1 is out of $0-$F range or if D0 is out of 0-6  ↩
   range.
Most of the time, you can choose any of those numbers $0-$F for D1
(which match trap numbers location $80 ->$BC) but some games use some traps, and
obviously you cannot use the same ones. You'd be very unlucky if the game used ALL ↩
    the traps.
In that case, you've got to patch manually (hard luck!)

A0: start address: MUST BE AN EVEN ADDRESS!
A1: end address
D0: register number (0-6)
D1: trap number ($0-$F)

This function flushes the caches at the end of the operation.

See also: WaitBlit()
```

1.104 StoreCopperPointer()

```
void StoreCopperPointer(cptr) - Logs the current (believed) copperlist pointer to  ↩
    JST
                          D0

D0 can be guessed or read from an instruction in the game.

Passing -1.L to this function will disable Iconification (JST believes the  ↩
    copperlist
is not known anymore)

See also: InGameIconify(), TellCopperPointer(), PatchMoveCList_Abs(),  ↩
    PatchMoveCList_Idx()
```

1.105 TellCopperPointer()

```
ULONG TellCopperPointer(void) - Returns the current logged copperlist pointer
 D0

Gets current copperlist pointer if you had logged the copperlist moves.

This function will return -1.L if the copperlist has not been logged yet.

See also: InGameIconify(), TellCopperPointer(), PatchMoveCList_Abs(),  ↩
    PatchMoveCList_Idx()
```

1.106 BeamDelay()

```
void BeamDelay(beamticks) - waits using VPOSR register
                  D0.W

This function will wait approx. D0*20ms.
Use it to fix some CPU dependent loops like

loop
  dbf Dx,loop

by dividing Dx by #$28 and calling BeamDelay
with value of Dx in D0
(problem mentionned by Harry in some Soundtracker replay routines)

  move.w  Dx,D0
  divu  #$28,D0
  swap  D0
  clr.w D0
  swap  D0
  JSRGEN  BeamDelay
(or   BEAM_DELAY  D0)


See also: BEAM_DELAY (macro)
```

1.107 WaitBlit()

```
void WaitBlit(void) - waits for blitter operations to complete

This small routine, inserted at the proper location, allows to avoid
blitter gfx bugs in loads of games (DoodleBug, Premiere, James Pond,
Ninja Spirit...)
and avoid crashes due to those blitter bugs (Lotus I, Lotus III, X-Out...)

You can either insert it before the blit (better) or after the blit
(safer, but can slowdown the game)

A blit is a write access to register $DFF058. See the JOTDHDInstall.guide
to learn how to find those faulty blits.

See also: WAIT_BLIT (macro)
```

1.108 SetQuitKey()

```
void SetQuitKey(key,user_routine) - Activates auto quit key
                D0       A0

D0: raw keycode
A0: user routine to be called before InGameExit

If A0=0, then no user routine will be called (default exit)

Note: Ralf prefered this feature to be reserved to registered users only. That's ↩
   his
will, and since he coded the stuff, I only have to agree.

If QUITKEY tooltype is set, SetQuitKey will use the QUITKEY tooltype value instead ↩
    of
the loader built-in value.

See also: SAVEOS_DATA (macro), SetIconifyKey()
```

1.109 SetIconifyKey()

```
ULONG SetIconifyKey(key,user_routine) - Activates auto iconify key
 D0                 D0       A0

D0: raw keycode (if D0 is 0 no iconify will be attempted)
A0: custom code which will be called:
   *before* InGameIconify with D0 = 0
   *after* InGameIconify with D0 = 1

If A0=0, then no user routine will be called.

If the Custom code returns with D0 != 0 no iconify will be performed
```

```
Remember that installing a iconify feature on a loader is tricky.:)

Note: Ralf prefered this feature to be reserved to registered users only. That's ↩
    his
will, and since he coded the stuff, I only have to agree. If used with a non- ↩
    registered
JST, this routine will do nothing.

If QUITKEY tooltype is set, SetQuitKey will use the QUITKEY tooltype value instead ↩
     of
the loader built-in value.

See also: InGameIconify(), SetQuitKey()
```

1.110 LogPatch()

```
                void LogPatch(address,length) - tells JST the patches you're going ↩
                    to do
                A0      D0

PATCHUSRJSR, PATCHUSRJMP, PATCH_RTS, PATCHGENJSR and PATCHGENJMP include this  ↩
    function
automatically if PATCH_LOGGED variable is set.

LogPatch saves D0 bytes of the memory at location A0. Useful to save the bytes you
patch upon.

The log file is called "patch.log". It is saved on exit in the current directory
or the directory specified by SAVEDIR. This logfile can be analysed using the

                printlog
                 tool.

See also: InitLogPatch, REGISTER_PATCH (macro)
```

1.111 InitLogPatch()

```
void InitLogPatch(void) - Initializes patch logging

Internal use. Called if the PATCH_LOGGED constant is defined in your loader BEFORE
the jst.i include.

See also: LogPatch, REGISTER_PATCH (macro)
```

1.112 The printlog tool

```
This tool is used to turn the binary patch.log files into viewable text.
This is very useful to update loaders to newer versions, which are generally
slightly different than the one you're already done...
```

```
Usage:

printlog <logfile> [<sourcefile>] [NOZPAGE]

logfile (compulsory): the logfile generated with JST (patch.log)
sourcefile (optionnal): put your asm source file here. When a patch address is  ↩
    found,
     printlog makes a "grep" in your sourcefile and displays the line used to patch ↩
         this location
     of course your sourcefile is NEVER written into.
NOZPAGE: does not show zero page patches matches in the sourcefile (if you patched ↩
     in $D0, you
     can have a trifle of lines matched :) )

Redirect the output to a file to see the patches you applied.

See also: LogPatch(), REGISTER_PATCH
```

1.113 Contact us

```
If you need any source code that is not in the current package, or you need
a precision on a function, or some help about doing something in an installer,
contact us:

Jean-François Fabre: jffabre@free.fr
Ralf Huvendiek: ralf@paderborn.netsurf.de
```