Default

Paul Manias

COLLABORATORS

	TITLE : Default		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Paul Manias	June 4, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Default

1.1 games.library

```
Name:          GAMES.LIBRARY AUTODOC
Version:       0.5 Beta.
Date:          15 February 1997
Author:        Paul Manias
Copyright:     DreamWorld Productions, 1996-1997.  All rights reserved.
Notes:         This  document is still being written and will contain errors
               in  a  number  of  places.   The information within cannot be
        treated as official until this autodoc reaches version 1.0.
```

1.2 Master Library Functions

```
 RemInputHandler()
 RemInterrupt()
 RemoveGPI()
 SetUserPrefs()
 SlowRandom()
 WaitTime()

 User Input Functions
 InitJoyPorts()
 ReadMouse()
 ReadJoyPort()
 ReadJoyStick()
 ReadJoyPad()
 ReadSegaPad()
 ReadAnalogue()
 ReadKey()
 WaitLMB()
 WaitFire()

 Data Processing Functions
 GetPicInfo()
 LoadPic()
 LoadPicInfo()
 QuickLoad()
 SmartLoad()
 SmartSave()
 SmartUnpack()
 UnpackPic()

 Object Processing Functions
 LoadObjectFile()
 FreeObjectFile()
 GetObject()
 GetObjectList()
```

1.3 Structure Layout

```
                             STRUCTURE LAYOUT


GMS  structures  have  been  designed  with just one commonality:  They all
start  with  a  version  header,  followed  by  a  private  "stats"  field.
Following this are whatever fields are relevant for that structure type.

Example:

     STRUCTURE  GameScreen,0
  LONG  GS_VERSION
  APTR  GS_Stats
  ...

The version header consists of a two character structure ID, followed by an
integer  that  usually  determines  the  version  number.   An  example for
GameScreens  is:   GSV1  = ("GS"<<16)|00.  The integer can be used for jump
tables  to  deal  with  the various structure types and handling the future
```

expansion of the structure.

The stats field follows immediately after the version ID, and is reserved for a second structure. This structure holds special information such as pre-calculated data for faster routines, and records of allocated memory. It is completely private, unless stated otherwise. If a structure is written to a file, then the stats field could contain the chunk size, as in IFF. To prevent confusion the Stats field must always be set to 0 when being initialised for the first time.

Structure IDentification can be used for more than tracking versions of passed structures. One such is example is the LIST ID header, which tells a function that it needs to perform the same action to more than one structure. You can see more about this in Lists.

STRUCTURE AUTO-INITIALISATION

A standard GMS policy for initialisation functions is to initialise all empty fields to either the user defaults, or values determined by any related fields. For example, omitting the ScrWidth and ScrHeight values from a screen would cause the screen to open at the user's ScrWidth and ScrHeight defaults. On the other hand if you were to omit the PicWidth and PicHeight settings, then these would inherit the values present in ScrWidth and ScrHeight. Sometimes if there is a file present, the values will come from that file's header structure. For example, IFF pictures will fill out a picture structure if it has empty fields.

The only fields that are not auto-initialised are the ones containing flags, such as the attrib and option fields.

FUTURE COMPATIBILITY

Structures are fully suppported as Data Objects. This means that you can still attain 100% future compatibility when initialising a pre-formatted structure (Tags do not even offer this level of compatibility). The only request is that your structures are located in an external OBJect file.

1.4 GMS Lists

LISTS

A list is intended for processing 2 or more structures inside a function. This is the fastest way that you can process a whole lot of structures without having to make heaps of function calls. Lets say you wanted to load in 10 sounds from your hard-drive using InitSound(). Normally InitSound() takes a Sound Structure, but it can also identify a list by checking the header ID.

To illustrate, a typical list for initialising/loading sounds looks like this:

```
SoundList:
  dc.l  "LIST"              ;List identification header.
  dc.l  SND_Boom            ;Pointers to each sound to load and
  dc.l  SND_Crash           ; initialise.
  dc.l  SND_Bang
  dc.l  SND_Ping
  dc.l  SND_Zoom
  dc.l  SND_Zig
  dc.l  SND_Zag
  dc.l  SND_Wang
  dc.l  SND_Whump
  dc.l  SND_Bong
  dc.l  LISTEND             ;Indicate an end to the list.
```

When you want to load all your sounds in, just use this piece of code:

```
  move.l  GMS_Base(pc),a6
  lea SoundList(pc),a0    ;a0 = Pointer to the soundlist.
  CALL  InitSound
  tst.l d0
  bne.s .error
```

Pretty easy right? Of course, there are lots of other functions that support lists. The not-so obvious ones are:

```
  InitBOB()
  InitSprite()
  InitSound()
  FreeSound()
```

Some functions are specially written to be given lists only, eg DrawBOBList(). This is mainly for speed reasons, as we don't want to waste time checking if a structure is a list or not in time critical situations.

That's basically the summary on lists. You may be interested to know that the GMS package is the only programmers aid that supports structures in this way. You will learn more about lists and how ID fields will help you in other areas of this doc.

1.5 Tags

GMS TAGS

GMS supports Tags in a way that is almost identical to the Amiga OS. The only major difference is that the new design allows them to operate a little faster. Tags allow you to support all future structure versions, and they are convenient for use in C. Unfortunately they take up more memory than a convential structure. Functions currently supporting tags are:

```
  AddScreen()
  LoadPic()
```

For C users the names of these functions are changed so that they have a "TAGS" suffix, eg AddScreenTags(). Assembler programmers can use the already existing functions. Note that tags are treated the same way as lists, and are correctly identified by functions only when they are passed a TAGS ID in the first field.

On the lowest level, tags are represented like this:

```
  dc.l  "TAGS",<Structure>
  dc.l  <ti_Tag>,<ti_Data>
  dc.l  TAGEND
```

Example:

```
  dc.l  "TAGS",GameScreen
  dc.l  GSA_ScrWidth,320
  dc.l  GSA_ScrHeight,256
  dc.l  TAGEND
```

If you omit the Structure and replace it with NULL, the relevant structure will be allocated for you. This structure will be placed in the NULL entry (useful for assembler programmers), and also be returned by the function. If a Tag call results in a return of NULL then an error has occured and the call has failed.

Here is an example of using Tags in C:

```
      struct GameScreen *GameScreen;

      if (GameScreen = AddScreenTags(TAGS,NULL,
           GSA_Planes,AMT_PLANES,
           GSA_Palette,Palette,
           GSA_ScrMode,LORES|COL24BIT,
           GSA_ScrWidth,320,
           GSA_ScrHeight,256,
           GSA_ScrType,INTERLEAVED,
           GSA_ScrAttrib,DBLBUFFER,
           TAGEND)) {

         /* Code Here */

      DeleteScreen(GameScreen);
      }
```

There are also some special flags that you can use for advanced Tag handling. These flags are identified in ti_Tag, and they are:

TAG_IGNORE - Skips to the next Tag entry.

TAG_MORE - Terminates the current TagList and starts another one (pointed to in the ti_Data field).

TAG_SKIP - Skips this and the next ti_Data items.

That's all you need to know, just remember to terminate all your tag calls with TAGEND.

1.6 GMS Error Codes

ERROR CODES

GMS has a universal set of error codes that are used by functions with a return type of ErrorCode. This enables you to easily identify errors and debug these problems when they occur. Here is a description of current error codes and what they mean:

[0] ERR_OK
No error occurred, function has executed successfully.

[1] ERR_NOMEM
Not enough memory was available when this function attempted to allocate a memory block.

[2] ERR_NOPTR
A required structure address pointer was not present.

[3] ERR_INUSE
This structure has previous allocations that have not been freed.

[4] ERR_STRUCT
You have given this function a structure version that is not supported, or you have passed it an unidentifiable memory address.

[5] ERR_FAILED
An unspecified failure has occurred.

[6] ERR_FILE
Unspecified file error, eg file not found, disk full etc.

[7] ERR_DATA
This function encountered some data that has unrecoverable errors.

[8] ERR_SEARCH
An internal search was performed and it failed. This is a specific error that can occur when the function is searching inside file headers for something, eg the BODY section of an IFF file.

[9] ERR_SCRTYPE
Screen Type not recognised or supported, eg currently True Colour modes are not available.

[10] ERR_GPI
This function tried to initialise a GPI and failed.

1.7 GMS Data Objects

GMS DATA OBJECTS

One of the problems with conventional games programming is that after the game has been compiled, all the structures and object data is often fixed in place, impossible to edit from a user point of view, and has no potential of future expansion.

By providing support for external data objects, we can achieve the possibility of up to 100% of data editing with very little effort. This opens up a large number of avenues for the future of your product. Even if you stop developing it, other users can still make improvements. For example:

Graphic Artists may edit your graphics in all areas, such as upgrading them to 24bit quality, changing resolutions from 320x256 to 1280x1024, altering the size, amount of animation frames, and clipping of your BOBs, adding and changing RasterList commands, and so on.

Programmers may change existing code segments to create new effects, improve compatibility, make time critical sections faster, and generally change whatever you allow them to.

Game Players could design new levels, change attack plans, game settings, and edit the game to suit their own tastes.

The File Format
Data Objects are compiled into a single binary file. The easiest way to learn how it works is to view one; here is an example of a GameScreen and a picture located in an object file:

```
                                ---START---

  ORG $0      ;Data is absolute.

  ;All object files start with "GOBJ" and then the data objects start
  ;immediately after this.

  dc.l  "GOBJ"      ;File identification.

  ;The GameScreen object starts with the compulsory object header,
  ;which also contains the name of the object in question.  You need
  ;to remember the names of all your objects as this is the only way
  ;to correctly identify them.  The structure data then follows in
  ;the data section

OBJ_GameScreen:
  dc.l  "STRC"      ;Object is a STRC [Structure].
  dc.l  .end      ;Pointer to the next structure.
  dc.b  "DemoScreen",0    ;Name.
  even
.data dc.l  GSV1,0
  dc.l  0,0,0     ;Screen memory 1/2/3.
  dc.l  0     ;Screen link.
  dc.l  0     ;Address of palette.
  dc.l  0     ;Address of rasterlist.
  dc.l  0     ;Amount of colours in palette.
  dc.w  640,256     ;Screen Width and Height.
  dc.w  0,0,0     ;Picture Widths and Height.
  dc.w  0     ;Amount of planes.
  dc.w  0,0     ;X/Y screen offset.
  dc.w  0,0     ;X/Y picture offset.
  dc.l  CENTRE      ;Special attributes.
  dc.w  0     ;Screen mode.
  dc.w  0     ;Screen type
.end

  ;The layout of the Picture object is generally identical to the
  ;GameScreen, we have just changed the name and entered the
  ;correct structure data.

OBJ_Picture:
  dc.l  "STRC"      ;Object is a STRC [Structure].
  dc.l  .end      ;Pointer to the next structure.
  dc.b  "DemoPicture",0   ;Name.
  even
.data dc.l  PCV1,0      ;Version header.
  dc.l  0     ;Source data.
  dc.w  640,0,256   ;Width, Height.
  dc.w  4     ;Amount of Planes.
  dc.l  16      ;Amount of colours.
  dc.l  0     ;Source palette.
  dc.w  HIRES|COL12BIT|LACED  ;Screen mode.
  dc.w  ILBM      ;Screen type.
  dc.l  GETPALETTE|VIDEOMEM ;Parameters.
  dc.l  .file
.file dc.b  "GAMESLIB:data/IFF.Pic640x256",0
  even
.end
  ;All files must terminate with an OEND string.
  dc.l  "OEND"
                                ---END---
```

In time there will be an editor for object files, so everyone will be able to create and edit them in a GUI interface rather than with an assembler.

GRABBING DATA FROM OBJECT FILES

You can grab a pointer to an object by first loading in the file, then using the GetObject(), GetObjectList() or CopyObject() functions. All you need to do is supply the name of the object you wish to grab and the function will find it for you.

If you want to find more than one object, you can use an object list. This is a special list designed for the GetObjectList() function. It looks like this:

```
  dc.l  "OLST"
  dc.l  <Name>,<Object>
  dc.l  ...
  dc.l  LISTEND
```

<Name> points to the name of the object you wish to find. <Object> will be initialised by the GetObjectList() function, ie it will point to the object if it finds it. Normally you will set this field as NULL before calling the function, if you place something in this field then GetObjectList()

```
will ignore that particular entry.

You  may  also mix different kinds of objects in the same list, eg BOBs and
Sounds can all be found in one call.

Generally  all  of  the Init() functions (eg InitBOB()) will support object
lists  if  they  are  supplied  with  one.  These functions will ignore any
structures  that  they  do  not recognise, eg InitBOB() will not attempt to
initialise sound samples, so it is safe for different structures to be
mixed into one list.
```

1.8 games.library/InitGPI

```
games.library/InitGPI

NAME  InitGPI - Load in a GPI and initialise it for function calls.

SYNOPSIS
  GPIBase = InitGPI (GPINumber, Version).
          d0                  d0        d1

  APTR InitGPI(UWORD GPINumber, UWORD Version);

FUNCTION
  Loads  in  a  GPI  and  initialises  it  ready  for function calls.
  Currently  there are three GPI's that require initialisation if you
  want to use them:

  Debug.GPI
  Network.GPI
  Vectors.GPI

  If  GPIBase returns with an address pointer then the initialisation
  was  successful  and  the GPI's functions are ready to use.  If the
  function fails then it will return with NULL.

NOTE  The GPIBase is the same as a library base pointer.  Because of this
  it  is  perfectly  legal  to  make  direct calls to the GPI itself.
  However, do not make direct calls to the Sound, Screens and Blitter
  GPI's as they do expect to be called with the games.library base in
  register a6.

  As  the  Debug,  Network and Vector GPI's are not present yet, this
  function is a bit useless for the moment :-)

INPUTS  GPINumber - A recognised GPI ID Number, which is one of:

  GPI_SCREENS, GPI_BLITTER, GPI_SOUND, GPI_NETWORK, GPI_VECTORS,
  GPI_DEBUG,   GPI_ANIM,    GPI_REKO,  GPI_TEXT.

  Version - The minimum GPI version that you require.

RESULT  GPIBase - Pointer to the GPIBase or NULL if error.

SEE ALSO
```

```
  RemoveGPI
```

1.9 games.library/RemoveGPI

```
games.library/RemoveGPI

NAME  RemoveGPI -- Remove a GPI that was previously initialised.

SYNOPSIS
  RemoveGPI(GPIBase)
                  a0

  ULONG RemoveGPI(APTR GPIBase);

FUNCTION
  Informs  the  games.library  that  you  no  longer  wish to use the
  specified  GPI's  functions.   You cannot make any calls to the GPI
  after removing it.

  All GPI's that you open must be removed before your program exits.

INPUTS  GPIBase - Pointer to a valid GPIBase returned from InitGPI().

SEE ALSO
  InitGPI
```

1.10 games.library/InitJoyPorts

```
games.library/InitJoyPorts

NAME  InitJoyPorts -- Initialise  the  JoyPorts  and  reset  the movement
        counters.

SYNOPSIS
  InitJoyPorts()

  void InitJoyPorts(void)

FUNCTION
  If  you  are  using  any of the JoyPort related functions, then you
  will  have  to initialise the ports before trying to use them.  You
  must  call  this  function  in  the  initialisation section of your
  program,  after  you  have called AddInputHandler() (or AddScreen()
  which will do this for you).

  You  will  also need to call this function if you need the movement
  counters  reset  (note  that  even  when  you  are  not reading the
  joyports  an interrupt will be keeping track of any change in their
  movements).   If  the user was to move an input device when you are
  not  calling any Read function, a nonsense value may be returned if
  you start reading the ports again.
```

```
SEE ALSO
  ReadJoyPort
```

1.11 games.library/ReadMouse

```
games.library/ReadMouse

NAME  ReadMouse -- Gets the current mouse co-ordinates and button states.

SYNOPSIS
  ZBXY = ReadMouse(PortName)
         d0                 d0

  ULONG ReadMouse(UWORD PortName);

FUNCTION
  Reads  the  mouse port and returns any changes in its co-ordinates.
  The status of the mouse is returned in ZXBYStatus (a packed state).
  If  the user was not using the mouse, then ZBXYStatus will return a
  NULL value.

  If  you  do  not  call InitJoyPorts() at the start of your program,
  this function may may return nonsense values in the X/Y directions.
  Also  make  sure that you call InitJoyPorts() whenever you need the
  X/Y coordinate changes reset.

  This function also requires that the input handler has already been
  installed by GMS (Calling ShowScreen() will do this for you).

  JoyPorts 3 and 4 are not supported by this function.

EXAMPLE If you are having trouble unpacking the ZBXYStatus value in C, here
  is some code to get the X, Y and Z values.

        XPos += (BYTE)(ZBXY>>8);
        YPos += (BYTE)ZBXY;
        ZPos += (BYTE)(ZBXY>>24);

  To read the left mouse button:

  if (ZBXY&MB_LMB) {
     /* LeftMouse pushed... */
  }

INPUT PortName = JPORT1 or JPORT2.

RESULT  ZBXY - Contains changes in direction and button states.

        BYTE | BIT RANGE | DATA
  -----+-----------+-------------------------------------
    1  |  0 -  7   | Y Direction
    2  |  8 - 15   | X Direction
    3  | 16 - 23   | Button status bits.
    4  | 23 - 31   | Z Direction (currently not supported)
  Button status bits are:

  MB_LMB - Left mouse button
  MB_RMB - Right mouse button
  MB_MMB - Middle mouse button

SEE ALSO
  games/gamesbase.i
```

1.12 games.library/ReadJoyPort

```
games.library/ReadJoyPort

NAME  ReadJoyPort -- Reads any joystick device in a given joyport.

SYNOPSIS
  JoyStatus = ReadJoyPort(PortName, ReturnType)
      d0                    d0         d1

  ULONG ReadJoyPort(UWORD PortName, UWORD ReturnType)

FUNCTION
  Reads  the  joyport  and returns its status in the required format,
  regardless  of  what  playing  device  is  plugged  in.   Currently
  supported   devices   are  standard  JoySticks, Analogue JoySticks,
  SegaPads, CD32 JoyPads, the mouse, and the keyboard.

  Unlike  the  lowlevel.library  equivalent  of  this  function, this
  version is much faster and does not need to evaluate what device is
  currently  plugged in.  It simply reads the specified joy type from
  GMSPrefs and jumps to the correct routine.

  Future  devices  may  be  added  to  this  function  - this will be
  transparent to your program so that you can support devices that do
  not exist yet.

NOTE  The  first  time  you  call  this  function  it may return nonsense
  values.  Therefore you must call InitJoyPorts() before use.

INPUTS  PortName - JPORT1, JPORT2, JPORT3 or JPORT4.
  ReturnType - JT_SWITCH: JoyStatus returns with switched bitflags.
               JT_ZBXY:   JoyStatus returns with the ZBXY format.

RESULT  JoyStatus - Status  of  the  JoyPort  in  one  of the following two
    formats:

  For JT_SWITCH you will be returned the joyport status in bits which
  are set by:

  JS_LEFT, JS_RIGHT, JS_UP, JS_DOWN, JS_ZIN, JS_ZOUT, JS_FIRE1, JS_FIRE2,
  JS_PLAY, JS_RWD, JS_FFW, JS_GREEN, JS_YELLOW.

  For  JT_ZBXY  you  will  be returned the joyport status in a packed
  state, containing directional values and button status bits:
        BYTE | BIT RANGE | DATA
  -----+-----------+-------------------------------------
    1  |  0 -  7   | Y Direction
    2  |  8 - 15   | X Direction
    3  | 16 - 23   | Button status bits.
    4  | 23 - 31   | Z Direction (currently not supported)

  Button bits: JB_FIRE1/MB_LMB, JB_FIRE2/MB_RMB, JB_FIRE3/MB_MMB.

SEE ALSO
  ReadMouse, ReadJoyStick, ReadJoyPad, ReadSegaPad, ReadAnalogue,
  games/games.i
```

1.13 games.library/ReadJoyStick

```
games.library/ReadJoyStick

NAME  ReadJoyStick -- Read the joystick status from a given joyport.

SYNOPSIS
  JoyBits = ReadJoyStick(PortName)
         d0                      d0

  ULONG ReadJoyStick(UWORD Portname);

FUNCTION
  Interprets  the  current  status  of  a joystick in the given port.
  Ports  3 and 4 are recognised as extended joysticks in the parallel
  port.   If  the  user was not using the joystick, then JoyBits will
  return a NULL value.

NOTE  Try  to  use  ReadJoyPort(),  as  that gives the same results, but
  supports Joypads, Analogue joysticks etc.

INPUTS  PortName - JPORT1, JPORT2, JPORT3 or JPORT4.

RESULT  JoyBits - The  current joystick status bits.  These are:

  JS_LEFT   =  0
  JS_RIGHT  =  1
  JS_UP     =  2
  JS_DOWN   =  3
  JS_FIRE1  =  6
  JS_FIRE2  =  7
  JS_FIRE3  =  8

SEE ALSO
  ReadJoyPort, ReadJoyPad, ReadSegaPad, ReadAnalogue,
  games/games.i
```

1.14 games.library/ReadAnalogue

```
games.library/ReadAnalogue

NAME  ReadAnalogue -- Read an analogue joystick from the given port.

SYNOPSIS
        ZBXYStatus = ReadAnalogue(PortName)
            d0                        d0

  ULONG ReadAnalogue(UWORD PortName);

FUNCTION
  Reads  an analogue joystick in either port 1 or port 2.  The status
  of the joystick is returned in ZXBYStatus (a packed state).  If the
  user was not using the joystick, then ZBXYStatus will return a NULL
  value.

  The first time you call this function it may return nonsense values
  in  the  X/Y  directions.   Therefore  you  must  call  it  in  the
  initialisation  section of your program before using it in the rest
  of your program.

  JoyPorts 3 and 4 are not supported by this function.

EXAMPLE If you are having trouble unpacking the ZBXYStatus value in C, here
  is some code to get the X, Y and Z values.

        XPos += (BYTE)(ZBXY>>8);
        YPos += (BYTE)ZBXY;
        ZPos += (BYTE)(ZBXY>>24);

INPUTS  PortName - JPORT1 or JPORT2.

RESULT  ZBXYStatus - Current status of the analogue joystick.

  The status data looks like this:

        BYTE | BIT RANGE | DATA
  -----+-----------+-------------------------------------
    1  |  0 -  7   | Y Direction
    2  |  8 - 15   | X Direction
    3  | 16 - 23   | Button status bits.
    4  | 23 - 31   | Z Direction (currently not supported)

  Note  that the further the joystick is pushed in a given direction,
  the  higher  the  value  returned  for the relevant byte.  Negative
  values denote a push in the opposite direction.

BUGS  NOT IMPLEMENTED YET.

SEE ALSO
  ReadJoyPort, ReadJoyStick, ReadSegaPad, ReadJoyPad,
  games/games.i
```

1.15 games.library/ReadJoyPad

```
games.library/ReadJoyPad

NAME  ReadJoyPad -- Reads a CD32 joypad from a specified port number.

SYNOPSIS
  JoyBits = ReadJoyPad(PortName)
          d0                    d0

  ULONG ReadJoyPad(UWORD PortName);

FUNCTION
  Reads  a  standard  Amiga JoyPad (ie a CD32 joypad) and returns its
  current  status  in  the JoyBits format.  If the user was not using
  the joypad, then JoyBits will return a NULL value.

INPUTS  PortName - JPORT1 or JPORT2.

RESULT  JoyBits - Current joypad status bits. These are:

  JS_LEFT   = 0
  JS_RIGHT  = 1
  JS_UP     = 2
  JS_DOWN   = 3
  JS_RED    = 6
  JS_BLUE   = 7
  JS_PLAY   = 8
  JS_RWD    = 9
  JS_FFW    = 10
  JS_GREEN  = 11
  JS_YELLOW = 12

  The red and blue buttons are the equivalent of fire buttons 1 and 2
  on a standard joystick.

BUGS  I have not tested this!

SEE ALSO
  ReadJoyPort, ReadJoyStick, ReadSegaPad, ReadAnalogue, games/games.i
```

1.16 games.library/ReadSegaPad

```
games.library/ReadSegaPad

NAME  ReadSegaPad - Reads a Sega joypad from a specified port number.

SYNOPSIS
  JoyBits = ReadSegaPad(PortName)
     d0      d0

  ULONG ReadSegaPad(UWORD PortName)

FUNCTION
```

```
  Reads  a standard Sega JoyPad and returns its current status in the
  JoyBits  format.   If  the  user  was  not  using the SegaPad, then
  JoyBits will return a NULL value.

INPUTS  PortName - JPORT1 or JPORT2.

RESULT  JoyBits - Current joypad status bits. The flags are:

  JS_LEFT, JS_RIGHT, JS_UP, JS_DOWN, JS_FIRE1, JS_FIRE2

BUGS  This has not even been tested by me!  Somone test it and tell me if
  it works OK.

SEE ALSO
  ReadJoyPort, ReadJoyStick, ReadJoyPad, ReadAnalogue, games/games.i
```

1.17 games.library/ReadKey

```
games.library/ReadKey

NAME  ReadKey -- Reads the keyboard and returns any new keypresses.

SYNOPSIS
  KeyValue = ReadKey(Keys)
          d0        a1

  UBYTE ReadKey(struct Keys *);

FUNCTION
  Checks  to  see  if  there  was  a keypress since the last time you
  called  this  routine.   If  there were no keypresses then KeyValue
  will return a NULL value.

  Most  key values are returned as ANSI, which is of the range 1-127.
  Special  keys  (eg  Cursor Keys, function Keys etc) are held in the
  range  of  128-255.   You  can  see  what these special keys are in
  games.i.

  Qualifiers  have  automatic  effects  on the ANSI value (eg shift+c
  will  return  "C").   Alt  keys,  Ctrl keys, and Amiga keys have no
  effect on the ANSI value.

  The  KeyStruct  is  also updated for future reference.  A KeyStruct
  will  hold  up  to four keys since your previous check.  If you are
  calling ReadKey() every vertical blank, you are already supporting
  typing  speeds  of  an  astronomical 600 words per minute, so it is
  only  necessary  to  check  KP_Key1.  If you are only grabbing keys
  every 1/2 second, then all fields should be checked.

NOTE  The GMS input handler needs to be active for this function to work.
  This  is  done  by calling ShowScreen() or AddInputHandler() in the
  initialisation section of your program.

INPUT Keys - Pointer to a valid Keys structure.  This structure is in the
         form of:
```

```
        STRUCTURE  KP,00
      UWORD  KP_ID                ;Updated by function, ignore.
      UBYTE  KP_Key1              ;Newest KeyPress.
      UBYTE  KP_Key2              ;...
      UBYTE  KP_Key3              ;...
      UBYTE  KP_Key4              ;Oldest KeyPress.

RESULT  KeyValue - Contains  the  latest keypress value, ie is identical to
       KP_Key1.
  Keys     - Updated  to  hold new key data.  You may receive as much
       as 4 keys in the provided fields.  Key fields containing
       zero indicate that no key was pressed.

SEE ALSO
  AddInputHandler, games/misc.i
```

1.18 games.library/FastRandom

```
games.library/FastRandom

NAME  FastRandom -- Generate a random number between 0 and <Range>.

SYNOPSIS
        Random = FastRandom(Range)
          d0          d1

  UWORD FastRandom(UWORD Range);

FUNCTION
  Creates  a  random number as quickly as possible.  The routine uses
  one divide to determine the range and will automatically change the
  random seed value each time you call it.  This routine has now been
  fully tested and generates 100% patternless numbers.

  Remember that all generated numbers fall BELOW the Range.  Add 1 to
  your range if you want this number included.

INPUTS  Range - A  range  between  1 and 32767.  An invalid range of 0 will
    result in a division by zero error.

RESULT  Random - A number greater or equal to 0, and less than Range.

SEE ALSO
  SlowRandom, demos/randomplot
```

1.19 games.library/SlowRandom

```
games.library/SlowRandom

NAME  SlowRandom -- Generate a random number between 0 and <Range>.
```

```
SYNOPSIS
  Random = SlowRandom(Range)
    d0          d1

  ULONG SlowRandom(UWORD Range);

FUNCTION
  Generates a very good random number in a relatively short amount of
  time.  This  routine  takes  approximately  two  times longer than
  FastRandom(),  but  is guaranteed of giving excellent random number
  sequences.

  Remember that all generated numbers fall BELOW the Range.  Add 1 to
  your range if you want this number included.

INPUTS  Range - A range between 1 and 32767.

RESULT  Random - A number greater or equal to 0, and less than Range.

SEE ALSO
  FastRandom, demos/randomplot
```

1.20 games.library/WaitLMB

```
games.library/WaitLMB

NAME  WaitLMB -- Wait for the user to hit the left mouse button.

SYNOPSIS
  WaitLMB()

  void WaitLMB(void);

FUNCTION
  Waits  for  the  user  to  hit  the left mouse button.  It will not
  return to your program until this event occurs.  Multi-tasking time
  will  be increased while waiting and an implanted AutoSwitch() call
  supports screen switching.

SEE ALSO
  ReadMouse, WaitFire
```

1.21 games.library/WaitFire

```
games.library/WaitFire

NAME  WaitFire -- Wait for the user to hit a fire button.

SYNOPSIS
  WaitFire(PortName)
        d0
  void WaitFire(UWORD PortName);

FUNCTION
  Waits  for  the user to hit the fire button.  It will not return to
  your  program  until this event occurs.  Multi-tasking time will be
  increased while waiting and an implanted AutoSwitch() call supports
  screen switching.

INPUTS  PortName - JPORT1, JPORT2, JPORT3 or JPORT4.

SEE ALSO
  ReadJoyStick, ReadJoyPad, ReadSegaPad, WaitLMB, games/games.i
```

1.22 games.library/WaitTime

```
games.library/WaitTime

NAME  WaitTime -- Wait for a specified amount of micro-seconds.

SYNOPSIS
  WaitTime(MicroSeconds)
          d0

  void  WaitTime(UWORD MicroSeconds);

FUNCTION
  Waits for a specified amount of micro-seconds.  During this time it
  will   reduce   the   task  priority  and  make  regular  calls  to
  AutoSwitch() for you.

INPUT MicroSeconds - Amount of micro-seconds to wait for.
```

1.23 games.library/AddInputHandler

```
games.library/AddInputHandler

NAME  AddInputHandler -- Add an input handler to the system.

SYNOPSIS
  AddInputHandler()

  void AddInputHandler(void)

FUNCTION
  Adds  an input handler at the highest priority to delete all system
  input  events.   The  idea  behind this is to prevent input falling
  through  to system screens and to give you more CPU time by killing
  all inputs.

  If  you  are going to use any of the Read functions (eg ReadKey())
  then  it  is  vital  that this function is active.  This is because
  some  of  the  Read  functions  are  hooked  into the input handler
  that this function provides.

NOTE  By  default  this  function  is  always  called  by  ShowScreen().
  Therefore  you only need to call this routine if you are using some
  other screen opening routine not in the games.library.

SEE ALSO
  RemInputHandler
```

1.24 games.library/RemInputHandler

```
games.library/RemInputHandler

NAME  RemInputHandler -- Remove the active input handler.

SYNOPSIS
  RemInputHandler()

  void RemInputHandler(void)

FUNCTION
  Removes the active input handler from the system.  As a result this
  will also deactivate certain Read functions (eg ReadKey()).

NOTE  DeleteScreen()  automatically calls this function so that any input
  handlers set up by ShowScreen() are removed.

SEE ALSO
  AddInputHandler
```

1.25 games.library/AddInterrupt

```
games.library/AddInterrupt

NAME  AddInterrupt -- Activate a custom written hardware interrupt.

SYNOPSIS
  IntBase = AddInterrupt(Interrupt, IntNum, IntPri)
     d0                        a0        d0      d1

  ULONG AddInterrupt(APTR Interrupt, UWORD IntNum, BYTE IntPri)

FUNCTION
  Initialises  a  system-friendly hardware interrupt and activates it
  immediately.   See  the  SetIntVector() and AddIntServer() descrip-
  tions in the exec.library for more details on system interrupts.

INPUTS  Interrupt - Pointer to your interrupt routine.
  IntNum    - The hardware interrupt bit.
  IntPri    - The priority of the interrupt, -126 to +127.

RESULT  IntBase - Pointer  to  the  interrupt  base,  you have to save this
```

```
      address  and  pass it back to RemInterrupt() before your
      program exits.

SEE ALSO
  RemInterrupt, exec/SetVector, games/misc.i
```

1.26 games.library/RemInterrupt

```
games.library/RemInterrupt

NAME  RemInterrupt -- Remove an active interrupt.

SYNOPSIS
  RemInterrupt(IntBase)
                d0

  void RemInterrupt(ULONG IntBase)

FUNCTION
  Disable  and  remove  an  active  interrupt  from the system.  This
  function is identical to RemIntServer() in the exec.library, but is
  a little easier to handle.

INPUT IntBase - Pointer to an interrupt base returned from AddInterrupt().

SEE ALSO
  AddInterrupt, games/games.i
```

1.27 games.library/SmartLoad

```
games.library/SmartLoad

NAME  SmartLoad -- Load in a file and depack it if possible.

SYNOPSIS
  MemLocation = SmartLoad(FileName, Destination, MemType)
      d0                      a0          a1        d0

  ULONG SmartLoad(char *FileName, APTR Destination, ULONG MemType)

FUNCTION
  Loads  in  a  file  and  depacks  it if necessary.  If the function
  cannot  find  a  recognised packer for the file then it will assume
  that it is not packed, and load it in without alteration.

  SmartLoad()  is  written  to  be  as  intelligent  as possible when
  loading  the  file.   This  includes keeping memory usage as low as
  possible,  and  searching  the  current directory for a file if any
  disk  assignment  cannot be found.  Future revisions of SmartLoad()
  are likely to contain more of these types of intelligent features.

  Currently supported packers are XPK (external), PowerPacker (inter-
  nal)  and  RNC (internal).  The recommended packing method for your
  files  is  the  traditional  RNC packer, which does not require any
  extra buffers for unpacking.

  Files  packed  with  XPK  require  the  xpkmaster.library  and  the
  relevant  compressor  in  your  LIBS:  directory, if the file is to
  unpack.  Keep this in mind when distributing your game.

  If  you  pass  NULL  as  the  Destination address, SmartLoad() will
  allocate  the  memory for you and return it in MemLocation, but you
  must give a recognised memory type.

  If  you  give  the  Destination  for  the  file then the MemType is
  ignored.

NOTE  If  you  wanted  the  allocation  you  will  have  to  free it with
  FreeMemBlock() when you are finished with it.

INPUTS  FileName    - Pointer to a null terminated string containing a file
          name.
  Destination - Destination for unpacked data or NULL for allocation.
  MemType     - Memory Type (only required if Destination is NULL).

RESULT  MemLocation - Pointer to the loaded data or NULL if failure.

SEE ALSO
  QuickLoad, SmartUnpack, <exec/memory.i>
```

1.28 games.library/QuickLoad

```
games.library/QuickLoad

NAME  QuickLoad -- Load in a file without any depacking.

SYNOPSIS
  MemLocation = QuickLoad(FileName, Destination, MemType)
       d0                   a0          a1        d0

  APTR QuickLoad(char *FileName, APTR Destination, ULONG MemType)

FUNCTION
  Loads  in a file without attempting to depack it.  The advantage of
  this  function is that it will assess the file size and load it all
  in for you.  It can also allocate the memory space if required, and
  has  limited  directory  searching  as  in SmartLoad(), if the file
  cannot immediately be found.

  If  you  pass  NULL  as  the  Destination address, QuickLoad() will
  allocate the memory for you but you must supply a recognised memory
  type.  If you give the Destination for the file then the MemType is
  ignored.

NOTE  If  you  wanted  the  allocation  you  will  have  to  free it with
  FreeMemBlock() when you are finished with it.
```

```
INPUTS  FileName    - Pointer to a null terminated string containing a file
          name.
  Destination - Destination for unpacked data or NULL for allocation.
  MemType     - Memory Type (only required if Destination is NULL)

RESULT  MemLocation - Pointer to the loaded data or NULL if failure.

SEE ALSO
  SmartLoad, SmartUnpack
```

1.29 games.library/SmartUnpack

```
games.library/SmartUnpack

NAME  SmartUnpack -- Unpack data from one memory location to another.

SYNOPSIS
  MemLocation = SmartUnpack(Source, Destination, Password, MemType)
           d0          a0        a1          d0        d1

  APTR SmartUnpack(APTR Source, APTR Destination, ULONG Password,
        ULONG MemType)

FUNCTION
  Attempts  to unpack a data area if it can assess the packing method
  used.   The  data  should begin with an ID longword followed by the
  size  of  the  original data before it was packed.  The data itself
  must  follow directly after this.  Any packer that does not do this
  will not be supported by this function.

  If  you  pass  NULL  as the destination address, SmartUnpack() will
  allocate  the memory for you, but you must give a recognised memory
  type.  If you give the Destination, the MemType is ignored.

  This  function  currently  supports  XPK  (external)  and  the  RNC
  (internal)  packer  types.  The RNC packer can unpack directly over
  itself  (ie  Source  and  Destination can be the same).  Do not try
  this with the XPK packer - it won't work!

NOTE  Remember  to  free  any  memory  returned  in  MemLocation  with
  FreeMemBlock() if you wanted the allocation.

INPUTS  Source      - Pointer  to  start  of  packed  data  (must  be an ID
          header).
  Destination - Destination for unpacked data or NULL for allocation.
  Password    - FileKey or NULL if none is used.
  MemType     - Memory type (only supply if Destination is NULL).

RESULT  MemLocation - Pointer to the unpacked data.

SEE ALSO
  SmartLoad
```

1.30 games.library/SmartSave

```
games.library/SmartSave

NAME  SmartSave -- Save a file to disk using a packer algorithm.

SYNOPSIS
  ErrorCode = SmartSave(FileName, Source, SrcLength)
     d0                    a0       a1       d0

  UWORD SmartSave(char *FileName, APTR Source, ULONG SrcLength)

FUNCTION
  Packs  a file if possible, and then saves the resulting data out to
  disk.   The  currently  supported  packing  method is XPK-NUKE, but
  GMSPrefs will soon allow the user to select any XPK packing method.
  To  load  the data back into your game, you will have no choice but
  to use SmartLoad().

INPUTS  FileName  - Name of the file to save to.
  Source    - Pointer to the start of the source data.
  SrcLength - Amount of data to save.

RESULT  ErrorCode - A standard GMS errorcode.  NULL indicates success.

SEE ALSO
  SmartLoad, SmartUnpack, games/games.i
```

1.31 games.library/SetUserPrefs

```
games.library/SetUserPrefs

NAME  SetUserPrefs -- Initialise a new set of preferences.

SYNOPSIS
  ErrorCode = SetUserPrefs(Name)
     d0                    a0

  ULONG SetUserPrefs(char *Name)

FUNCTION
  Initialises  a  new  set  of GMS preferences for the games.library.
  The  function  will take the Name you have given and search for its
  directory  in  ENV:GMSPrefs/.   If  found,  the  settings  in  this
  directory  will  be loaded and each GPI will be reactivated for the
  new preferences to take effect.  If the Name is not found or if you
  supply a Name of NULL, the default settings will be loaded.  If the
  default  settings are not found, then the internal settings will be
  used.

  This  function may also set your tasks priority and perform various
  other   actions   that  can  directly  affect  your  task,  or  the
  environment  that  it  is  running  in.   For  this  reason,  it is
  essential  that  this  is  the  first  function that you call after
  opening the games.library.

  The prefences manager for altering game settings is GMSPrefs, which
  handles  all game directories, the default settings and so on.  For
  more information on the options available to the user, see the file
  GMSPrefs.guide.

NOTE  If ENV:GMSPrefs/ does not exist, ENVARC:GMSPrefs/ will be searched,
  then S:GMSPrefs/.

  The  field tc_UserData in your exec task node will be used to point
  to  a second GMSTask node.  If you need a UserData node, there is a
  link   called   gt_UserData   in   the   GMSTask   structure   (see
  games/tasks.i)  which you may use for your own means.  We recommend
  that  you  treat  this  field as a chain of links in case of future
  expansion.

INPUT Name - The name of the preferences directory to access, or NULL for
       the default.

RESULT  ErrorCode - Returns ERR_OK if successful.
```

1.32 games.library/LoadPic

```
games.library/Loadpic

NAME  LoadPic -- Load in a recognised picture file.

SYNOPSIS
  ErrorCode = LoadPic(Picture)
     d0                 a1

  Picture = LoadPic(TagList)
     d0       a1

  ULONG LoadPic(struct Picture *)

  struct Picture * LoadPicTags(unsigned long ...)

FUNCTION
  Loads  in  a  picture  file  (PIC_File), and if the picture type is
  recognised, unpacks the data to a buffer given in PIC_Data.  If you
  do  not  supply a data destination, then a buffer will be allocated
  for you and placed in PIC_Data.  If this is the case you will later
  have to call FreePic() to give this buffer back to the system.

  LoadPic() has all the standard features of GMS functions, including
  field initialisation for NULL fields.  Note that by setting certain
  fields  you  are  placing restrictions on the picture that is to be
  loaded.   For  example, if the picture is bigger then the specified
  width, the picture will have its right edge clipped.  To get around
  this  simply  leave  the  Width  field  at zero, and LoadPic() will
  initialise this field, loading the picture without clipping it.

NOTE  If  this  function cannot identify the source header, then the call
```

```
  will  fail.   Currently  the only supported format is IFF, but GIF,
  JPEG  and other picture format support will be added later (someone
  please send me the info!)

INPUT Picture - Pointer to a Picture structure or TagList.

  Here follows a description of each field:

  PIC_VERSION
  The version of the structure, currently PCV1.

  PIC_Data
  Pointer  to  the picture's data destination for the unpack.  If you
  specify  NULL  here, a buffer will be allocated and placed here for
  you.

  PIC_Width
  The  width of the picture in bytes.  This field will be initialised
  if  a  width  is  not  given  here.   Note that the picture will be
  clipped if it exceeds the width boundary.

  PIC_Height
  The   height  of  the  picture  in  pixels.   This  field  will  be
  initialised  if  a height is not given here.  Note that the picture
  will be clipped if it exceeds the height boundary.

  PIC_Planes
  The  amount  of  planes  in  this  picture.  As usual this field is
  initialised  if it is NULL.  Note that the picture will lose planes
  if it exceeds this value

  PIC_AmtColours
  The  amount  of  colours that you want to grab from the palette, or
  the  amount of colours available for the remap.  This field will be
  initialised if it is unspecified.

  PIC_Palette
  Points  to  a  palette if you want to use the REMAP option.  On the
  other hand if you specify the GETPALETTE option, then the picture's
  palette will calculated and placed in here.

  PIC_ScrMode
  The screen mode that this picture is being loaded into.  This field
  will be initialised for you if you specify GETVMODE in PIC_Options.
  Otherwise it is assumed you have filled out this field.

  PIC_Type
  The  data  type of this picture, PLANAR, INTERLEAVED or CHUNKY.  If
  you   omit  a  specification  in  this  field,  the  function  will
  initialise it to the user's preferred screen type.

  PIC_Options
  You  can  specify  certain  flags here that will affect the way the
  picture is initialised.  Valid flags are:

  GETPALETTE - Gets  the  palette of the picture and generates a copy
         of  the colour values in COL12BIT or COL24BIT formats.
```

```
        The  amount  of  colours  obtained is dependent on the
        PIC_AmtColours field.  If you specify 0 in that field,
        all the colours will be obtained.

  REMAP      - Remaps  the picture data to fit the palette pointed to
        in the PIC_Palette field.

  GETVMODE   - Gets the user's preferred screen mode and writes it to
        GS_ScrType.

  VIDEOMEM   - Allocates video memory that is displayable on screen.

  RESIZE     - Resizes   the  picture  so  that  it  fits  the  given
        dimension limits (PIC_Width, PIC_Height)

  PIC_File
  Pointer to a NULL terminated string, that contains the filename for
  this picture.  This field is ignored by the UnpackPic() function.

RESULT  ErrorCode - Returns NULL if successful.

SEE ALSO
  UnpackPic, FreePic, games/image.i
```

1.33 games.library/UnpackPic

```
games.library/UnpackPic

NAME  UnpackPic -- Unpack a picture to a designated buffer.

SYNOPSIS
  ErrorCode = UnpackPic(Source, Picture)
     d0                   a1      a0

  ULONG UnpackPic(APTR Source, struct Picture *)

FUNCTION
  Unpacks  the  data  contained in a recognised picture header to the
  data  destination  given  in PIC_Data.  If you do not supply a data
  destination,  then a buffer will be allocated for you and placed in
  PIC_Data.

  If  this  function cannot identify the source header, then the call
  will  fail.   The standard expected format is IFF, but GIF and JPEG
  support will be added (for benefit of the user) later.

INPUT Source - Pointer to the header of the picture source.
  Picture - Pointer to a Picture structure.

RESULT  ErrorCode - Returns NULL if successful.

SEE ALSO
  LoadPic, FreePic, games/image.i
```

1.34 games.library/GetPicInfo

```
games.library/GetPicInfo

NAME  GetPicInfo -- Get the information on a recognised picture type.

SYNOPSIS
  ErrorCode = GetPicInfo(Picture)
     d0                    a1

  ULONG GetPicInfo(struct Picture *)

FUNCTION
  This  function  will load a picture's information header (unless it
  is  already  present in PIC_Header), and then fills out the Picture
  structure  according to the information that it finds.  Only fields
  that are set to NULL will be initialised, so preset fields will not
  be affected.

  You  will  need  to  use  some  special  options  provided  by  the
  PIC_Options  field  to  get  certain  information.  GETPALETTE will
  write out the picture's palette data to the address in PIC_Palette.
  If  PIC_Palette  is empty then the correct amount of memory will be
  allocated and placed in this field for you.  GETVMODE will find the
  picure's resolution and colour modes and write it to PIC_ScrMode.

  By  using  this  function  you  can find information on any picture
  format currently supported by GMS.  If the picture format cannot be
  assessed, then an error code of ERR_DATA will be returned.

NOTE  You  will have to call FreePic() if any memory was allocated by the
  GetPicInfo()  function  (eg  if  GETPALETTE was specified without a
  pointer in PIC_Palette).

INPUT Picture - Pointer to a Picture structure.

RESULT  ErrorCode - Returns NULL if successful.

SEE ALSO
  LoadPic
```

1.35 games.library/AllocMemBlock

```
games.library/AllocMemBlock

NAME  AllocMemBlock -- Allocate a new memory block.

SYNOPSIS
  MemBlock = AllocMemBlock(Size, MemType)
     d0                     d0     d1

  APTR AllocMemBlock(ULONG Size, ULONG MemType)

FUNCTION
```

```
  Allocates  a memory block from the system - this function is almost
  identical  to  AllocVec()  in  the  exec  library.   It exists here
  because  AllocVec()  is  only  available  on V36+ machines, plus it
  offers  some extra features available for debugging purposes.

  Header and Tail ID's are used to offer a security system similar to
  MungWall,  acting  as  cookies  on  the  header  and tail of memory
  blocks.   You  will  be  alerted  by FreeMemBlock() if the ID's are
  damaged.  This is a permanent debugging feature, so there is little
  need  to  run MungWall for debugging your programs.

  Resource  tracking  is  available,  so  you  will  be warned if you
  allocate  memory  and  forget to free it on exit (ie when you close
  the   games.library).    This   memory   will  be  freed  for  your
  convenience.

  By  default  all  GMS  memory is cleared before it is given to you.
  For simplicity there are only a few memory types:

    MEM_ANY
    MEM_VIDEO
    MEM_BLIT
    MEM_SOUND

  MEM_ANY is suitable for basic programming purposes, such as storing
  variables  and  running  code.   On  current  Amiga's this could be
  either chip or fast memory.

  MEM_VIDEO  is  for displaying graphics, and is also compatible with
  the  Blitter.GPI.

  MEM_BLIT  is  memory  that  is  compatible  with  the  Blitter.GPI.
  Currently this GPI only uses chip memory, but future versions could
  also  support  CPU  drawing  from fast if the graphic is located in
  that area.

  MEM_SOUND  is  memory  that is compatible with the Sound.GPI.  Like
  the  Blitter.GPI  only  chip  memory is currently supported, but in
  future sounds could be buffered in fast memory.

  You  may  also  use  the  MEM_PUBLIC flag if other programs will be
  accessing  your  memory.  This type of memory is not tracked, so it
  is legal to have a different program free such a memory block.

INPUT Size    - Size of the required memblock in bytes.
  MemType - The type of memory to allocate, eg MEM_VIDEO.

RESULT  MemBlock -  Pointer to the start of your allocated memblock or NULL
        if  failure.   If  the  allocation  was successful then
        -8(MemBlock)  will  contain  the size of your allocated
        memory.   You  can  read this value, but DON'T write to
        it!  You can also check for valid memory allocations by
        looking   at  the  ID  header.   "MEMH"  is  placed  at
        -12(MemBlock),  and  "MEMT" is placed at the end of the
        memory block.

SEE ALSO
```

```
  FreeMemBlock
```

1.36 games.library/FreeMemBlock

```
games.library/FreeMemBlock

NAME  FreeMemBlock -- Free a previously allocated mem block.

SYNOPSIS
  FreeMemBlock(MemBlock)
               a0

  void FreeMemBlock(APTR MemBlock)

FUNCTION
  Frees  a memory area allocated by AllocMemBlock(), AllocVideoMem(),
  AllocBlitMem(),  or  AllocSoundMem().  If the mem header or tail is
  missing,  then  it  is  assumed that something has written over the
  boundaries  of  your  memblock,  or  you  are  attempting to free a
  non-existant  allocation.  Normally  this  would  cause a complete
  system  crash, but instead we simply alert you to the fact, and you
  can continue on.

  Bear  in  mind  that  it  does pay to save your work and reset your
  machine  if  such a message appears, as it indicates that important
  memory data may have been destroyed.

NOTE  Never attempt to free the same MemBlock twice.

INPUT MemBlock - Points  to  the  start  of a memblock.  If NULL, then no
       action will be taken (function exits).

SEE ALSO
  AllocMemBlock
```

1.37 games.library/DecToText

```
games.library/DecToText

NAME  DecToText -- Outputs a Number as decimal formatted text.

SYNOPSIS
  Address = DecToText(Number, AmtDigits, Destination)
    d0           d0  d1     a0

  APTR DecToText(LONG Number, ULONG AmtDigits, char *Destination)

FUNCTION
  Takes  a  Number and outputs it to Destination as decimal formatted
  text.  AmtDigits defines the maximum amount of digits that you want
  to  be  written  out.   If  the number does not completely fill the
  given amount of digits, it will be trailed with leading zero's.  If
  the  AmtDigits  parameter  is  NULL, the number will be output with
  left  alignment,  (no  leading zero's).  Negative numbers get a '-'
  character put in front.

INPUTS  Number      - A number to convert to text.
  AmtDigits   - The amount of digits to write out, or NULL if you
          want left alignment with no trailing 0's.
  Destination - Memory location of where you want the numeric text to
          be written out.

RESULT  Address - The address where this function stopped writing out any
      characters.
```

1.38 games.library/LoadObjectFile

```
games.library/LoadObjectFile

NAME  LoadObjectFile -- Loads a valid object file and readies it for use.

SYNOPSIS
  ObjectBase = LoadObjectFile(FileName)
     d0            a0

  APTR LoadObjectFile(char *FileName)

FUNCTION
  Loads  in  an object file using the SmartLoad() function.  The file
  must  be a recognised object file with a "GOBJ" header in the first
  4 bytes.

  If  you  want  to create your own object files, read the section on
  Data Objects.

INPUTS  FileName - Indicates where to find the object file on disk.

RESULT  ObjectBase - Start of the object file.  Returns NULL if failed.

SEE ALSO
  FreeObjectFile
```

1.39 games.library/FreeObjectFile

```
games.library/FreeObjectFile

NAME  FreeObjectFile -- Frees a previous loaded object file.

SYNOPSIS
  FreeObjectFile(ObjectBase)
        a0

  void FreeObjectFile(APTR ObjectBase)
```

```
FUNCTION
  Frees an object file that has been loaded in with LoadObjectFile().
  Objects that are loaded in and are not freed may present you with a
  resource tracking error when you close the games.library.

INPUTS  ObjectBase -  Pointer   to   a  valid  ObjectBase  as  returned  by
          LoadObjectFile().

SEE ALSO
  LoadObjectFile
```

1.40 games.library/GetObject

```
games.library/GetObject

NAME  GetObject -- Finds an object by Name and returns it.

SYNOPSIS
  Object = GetObject(ObjectBase, Name)
    d0            a0 a1

FUNCTION
  This  function  finds  an  object by Name, and returns a pointer to
  that  object  inside  the  ObjectBase.  This function does not make
  copies of the object, so any changes you make will be affecting the
  original  object  data.   This  should  be fine for the majority of
  circumstances.

  If  the  object  is a code segment, you can execute it in assembler
  using these instructions:

    CALL  GetObject
    tst.l d0
    beq.s .error
    move.l  d0,a0
    jsr (a0)

  Structure  segments  immediately  start  with  the Structure ID (eg
  GSV1),  as  do  TagLists ("TAGS").  Data segments point directly to
  the beginning of the data in question.

INPUTS  ObjectBase - Valid ObjectBase as returned by LoadObjectFile().
  Name     - Pointer to the name of the object that you wish to find.

RESULT  Object - Pointer to the Object, or NULL if not found.

SEE ALSO
  GetObjectList
```

1.41 games.library/GetObjectList

```
games.library/GetObjectList

NAME  GetObjectList --

SYNOPSIS
  ErrorCode = GetObjectList(ObjectBase, ObjectList)
     d0           a0     a1

  ULONG GetObjectList(APTR ObjectBase, struct *ObjectList[])

FUNCTION
  This  function  acts  the same way as GetObject() but will grab the
  objects  from  a  list  and  process  them one by one.  This is the
  fastest and most convenient way to obtain a large set of objects.

  Here is the ObjectList format:

    dc.l  "OLST"
    dc.l  <Name>,<Object>   ;PE_Name, PE_Object.
    dc.l  ...
    dc.l  LISTEND

  <Name>  points  to  a  character  string  correctly  identifying an
  object,  and  <Object>  should be NULL as it will be initialised by
  this function.

INPUTS  ObjectBase - Valid ObjectBase as returned by LoadObjectFile().
  ObjectList - A list of objects to initialise.

RESULT  ErrorCode  - Returns ERR_OK if successful.
  ObjectList - Will be updated so that each PE_Object field points to
         the relevant object that was found.

SEE ALSO
  GetObject
```

1.42 games.library/CopyObject

```
games.library/CopyObject

NAME  CopyObject -- Make a copy of the object and return it.

SYNOPSIS

FUNCTION

INPUTS

RESULT

SEE ALSO
```

1.43 games.library/FindGMSTask

```
games.library/FindGMSTask

NAME  FindGMSTask -- Find the GMSTask node for the current task.

SYNOPSIS
  GMSTask = FindGMSTask()

  struct GMSTask * FindGMSTask(void)

FUNCTION
  This  function  will  supply you with a pointer to the GMSTask node
  for  the  task that called this function.  The GMSTask node is used
  for  storing  data  that  is  specific  to  your task - things like
  preference  settings for example.  Generally most of the fields are
  considered  private,  although  you  may  read  values from them if
  necessary.

  For the curious, it only takes 3 assembler instructions to grab the
  task node, so there is no time wasted in calling this function.

RESULT
  GMSTask - Points to the GMSTask node.

SEE ALSO
  games/tasks.i
```

1.44 games.library/

```
games.library/

NAME

SYNOPSIS

FUNCTION

INPUTS

RESULT

SEE ALSO
```