Default

Paul Manias

COLLABORATORS			
	TITLE : Default		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Paul Manias	July 26, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Default **1**

1.1 Master.GPI . . . 1
1.2 Master GPI Functions . . . 1
1.3 Structure Layout . . . 3
1.4 GMS Lists . . . 4
1.5 Tags . . . 5
1.6 GMS Error Codes . . . 6
1.7 GMS Data Objects . . . 8
1.8 Master.GPI/InitGPI . . . 10
1.9 Master.GPI/RemoveGPI . . . 11
1.10 Master.GPI/InitJoyPorts . . . 11
1.11 Master.GPI/ReadMouse . . . 12
1.12 Master.GPI/ReadJoyPort . . . 13
1.13 Master.GPI/ReadJoyStick . . . 14
1.14 Master.GPI/ReadAnalogue . . . 14
1.15 Master.GPI/ReadJoyPad . . . 15
1.16 Master.GPI/ReadSegaPad . . . 16
1.17 Master.GPI/ReadKey . . . 17
1.18 Master.GPI/FastRandom . . . 18
1.19 Master.GPI/SlowRandom . . . 18
1.20 Master.GPI/WaitLMB . . . 19
1.21 Master.GPI/WaitTime . . . 19
1.22 Master.GPI/AddInputHandler . . . 19
1.23 Master.GPI/RemInputHandler . . . 20
1.24 Master.GPI/AddInterrupt . . . 20
1.25 Master.GPI/RemInterrupt . . . 21
1.26 Master.GPI/SmartLoad . . . 21
1.27 Master.GPI/QuickLoad . . . 22
1.28 Master.GPI/SmartUnpack . . . 23
1.29 Master.GPI/SmartSave . . . 23

1.30 Master.GPI/SetUserPrefs . . . 24
1.31 Master.GPI/LoadPic . . . 25
1.32 Master.GPI/UnpackPic . . . 27
1.33 Master.GPI/GetPicInfo . . . 27
1.34 Master.GPI/AllocMemBlock . . . 28
1.35 Master.GPI/FreeMemBlock . . . 29
1.36 Master.GPI/WriteDec . . . 30
1.37 Master.GPI/FreeObjectFile . . . 30
1.38 Master.GPI/GetObject . . . 31
1.39 Master.GPI/GetObjectList . . . 31
1.40 Master.GPI/CopyObject . . . 32
1.41 Master.GPI/FindGMSTask . . . 32
1.42 Master.GPI/CloseGMS . . . 33
1.43 Master.GPI/DebugMessage . . . 33
1.44 Master.GPI/ErrorMessage . . . 34
1.45 Master.GPI/AddTrack . . . 35
1.46 Master.GPI/DeleteTrack . . . 36
1.47 Master.GPI/InitDestruct . . . 36
1.48 Master.GPI/SelfDestruct . . . 37
1.49 Master.GPI/GetPicture . . . 38
1.50 Master.GPI/GetStructure . . . 38
1.51 Master.GPI/InitTags . . . 39
1.52 Master.GPI/CloseFile . . . 39
1.53 Master.GPI/OpenFile . . . 40
1.54 Master.GPI/ReadFile . . . 41
1.55 Master.GPI/WriteFile . . . 42
1.56 Master.GPI/LoadObjectFile . . . 42
1.57 Master.GPI/StepBack . . . 43
1.58 Master.GPI/LoadPicFile . . . 43
1.59 Master.GPI/GMSForbid . . . 44
1.60 Master.GPI/GMSPermit . . . 44
1.61 Master.GPI/ . . . 45

Chapter 1

Default

1.1 Master.GPI

```
Name:          MASTER.GPI AUTODOC
Version:       0.6 Beta.
Date:          11 May 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.

 GENERAL INFORMATION
 Structures
 Lists
 Tags
 Data Objects
 Error Codes

 FUNCTIONS
 Master.GPI
 Screens.GPI
 Blitter.GPI
 Sound.GPI
```

1.2 Master GPI Functions

```
 MASTER.GPI
 AddInputHandler()
 AddInterrupt()
 AllocMemBlock()
 CloseGMS()
 FastRandom()
 FindGMSTask()
 FreeMemBlock()
 GMSForbid()
 GMSPermit()
 InitGPI()
```

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

Debugging Functions
DebugMessage()
ErrorMessage()
StepBack()

Data Processing Functions
GetStructure()
InitTags()
SmartUnpack()

File Functions
CloseFile()
OpenFile()
QuickLoad()
ReadFile()
SmartLoad()
SmartSave()
WriteFile()

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

Picture Functions
GetPicInfo()
GetPicture()
LoadPic()
LoadPicFile()
UnpackPic()

Resource Tracking Functions
AddTrack()
DeleteTrack()
InitDestruct()
SelfDestruct()

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

```
WaitLMB()
```

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 field, 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 read 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

In GMS it is illegal to define a structure in your code and compile it into the final binary. The only way you can legally obtain a structure is to call its related Get*() function (eg GetPicture()) or to use a tag list, which will return an initialised structure to you. This solves all future compatibility concerns in GMS.

There are some exceptions, eg the JoyData structure which is fairly basic and easily recognised by the ReadJoyPort() function.

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  GMSBase(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 practically identical to the Amiga OS. The only major difference is that an internal change in 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. Because pre-compiled structures are generally illegal in GMS, you have to use tags a lot. Make sure that you look at the demos so that you understand how to use them.

Here are just some of the functions that support tags:

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

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

On the lowest level, tags are represented like this:

```
  dc.l  TAGS_ID,<Structure>
  dc.l  <ti_Tag>,<ti_Data>
  dc.l  TAGEND
```

Example:

```
  dc.l  TAGS_GAMESCREEN,0
  dc.l  GSA_ScrWidth,320
  dc.l  GSA_ScrHeight,256
  dc.l  TAGEND
```

If you omit the Structure and replace it with NULL as in this example, the relevant structure will be allocated for you. The newly allocated structure will be placed in the NULL entry (useful for assembler programmers), and will 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. To find out why the failure occured you would have to use a GMS

debugger like IceBreaker.

Here is an example of using tags in C:

```
      struct GameScreen *GameScreen;

      if (GameScreen = AddScreenTags(TAGS_GAMESCREEN,NULL,
           GSA_Planes,5,
           GSA_Palette,Palette,
           GSA_ScrMode,LORES,
           GSA_ScrWidth,320,
           GSA_ScrHeight,256,
           GSA_ScrType,ILBM,
           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. ErrorCodes are sent to IceBreaker with full descriptions, so use this program for easy identification of errors. 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.

[11] ERR_RASTCOMMAND
Invalid raster command detected. Check your rasterlist for errors and make sure it terminates with a RASTEND.

[12] ERR_RASTERLIST
Complete rasterlist failure. You have tried to do something which is not possible under present conditions.

[13] ERR_NORASTER
No rasterlist was found in GS_Rasterlist.

[14] ERR_DISKFULL
Disk full error, time to get more space.

[15] ERR_FILEMISSING
File not found.

[16] ERR_WRONGVER
Wrong version or version not supported.

[17] ERR_MONITOR
Monitor driver not found or not supported.

[18] ERR_UNPACK
Problem with unpacking of data.

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 OBJECT FILE FORMAT

Data Objects are compiled into standard Amiga segmented files. 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:

```
  INCDIR  "INCLUDES:"
  INCLUDE "games/games.i"

  SECTION "Start",CODE

Start:  cmp.l #"GKEY",d0      ;d0 = Are we being called from GMS?
  bne.s .exit           ;>> = Called from DOS, exit safely.
  move.l  #Objects,d0     ;d0 = Return pointer to start of data.
  rts

.exit moveq #$00,d0
  rts

;---------------------------------------------------------------------------;

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

Objects:
```

```
  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  "TAGS"           ;Object ID.
  dc.l  OBJ_Picture      ;Pointer to next object.
  dc.b  "Screen",0       ;Name.
  even
.data dc.l  TAGS_GAMESCREEN,0
  dc.l  GSA_AmtColours,16
  dc.l  GSA_ScrWidth,640
  dc.l  GSA_ScrHeight,256
  dc.l  GSA_Planes,4
  dc.l  GSA_Attrib,CENTRE
  dc.l  GSA_ScrMode,HIRES|LACED
  dc.l  GSA_ScrType,ILBM
  dc.l  TAGEND

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

OBJ_Picture:
  dc.l  "TAGS"          ;Object ID.
  dc.l  End             ;Pointer to next object.
  dc.b  "Picture",0     ;Name.
  even
.data dc.l  TAGS_PICTURE,0
  dc.l  PCA_AmtColours,16
  dc.l  PCA_Width,640
  dc.l  PCA_Height,256
  dc.l  PCA_Planes,4
  dc.l  PCA_ScrMode,HIRES|LACED
  dc.l  PCA_ScrType,ILBM
  dc.l  PCA_Options,GETPALETTE|VIDEOMEM
  dc.l  PCA_File,.file
  dc.l  TAGEND

.file dc.b  "GMS:demos/data/IFF.Pic640x256",0
  even

  ;All lists must terminate with an OEND string.

End:  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() or GetObjectList() 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. Note that identifiable tag structures (eg TAGS_GAMESCREEN) will be preprocessed by the object processor, so in this case you would be returned a GameScreen structure that already contains the values from the tag list.

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  OBJECTLIST,0
  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 Master.GPI/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, ULONG Version);
```

FUNCTION

  Loads in a GPI and initialises it ready for function calls.

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 not work in the same way.

INPUTS GPINumber - A recognised GPI ID Number from games/games.i.
  Version   - The minimum GPI version that you require.

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

SEE ALSO
  RemoveGPI
```

1.9 Master.GPI/RemoveGPI

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

SYNOPSIS
  RemoveGPI(GPIBase)
                    a0

  ULONG RemoveGPI(APTR GPIBase);

FUNCTION
  Informs  GMS  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 Master.GPI/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 Master.GPI/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 Master.GPI/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.

  For  JT_DATA you will be returned a data structure which is derived
  from the JT_ZBXY type.  It looks like this:

      STRUCTURE JD,0
    WORD  JD_XChange
    WORD  JD_YChange
    WORD  JD_ZChange
    UWORD JD_Buttons
    LABEL JD_SIZEOF

  The JD_Buttons field has the flags JD_FIRE1 ... JD_FIRE8.

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

1.13 Master.GPI/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, JS_RIGHT, JS_UP, JS_DOWN, JS_FIRE1, JS_FIRE2, JS_FIRE3

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

1.14 Master.GPI/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 Master.GPI/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, JS_RIGHT, JS_UP,    JS_DOWN, JS_RED,  JS_BLUE, JS_PLAY,
        JS_RWD,  JS_FFW,   JS_GREEN, JS_YELLOW.

  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 Master.GPI/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 Master.GPI/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 Master.GPI/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 Master.GPI/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 Master.GPI/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
```

1.21 Master.GPI/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.
```

1.22 Master.GPI/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 GMS.

SEE ALSO
  RemInputHandler
```

1.23 Master.GPI/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
```

1.24 Master.GPI/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, games/misc.i
```

1.25 Master.GPI/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
```

1.26 Master.GPI/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>
```

1.27 Master.GPI/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
```

1.28 Master.GPI/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
```

1.29 Master.GPI/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
```

1.30 Master.GPI/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 preferences for a GMS task.  This function
  will  take  the Name you have given and search for its directory in
  GMS:Prefs/.   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 GMS.

  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  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.
```

1.31 Master.GPI/LoadPic

```
NAME  LoadPic -- Load in a recognised picture file.

SYNOPSIS
  Picture = LoadPic(Picture/TagList)
     d0                   a1

  struct Picture * 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.

  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 unspecified, and LoadPic() will
  initialise  this  field,  loading  the picture without clipping it.
  Alternatively  you  could specify the RESIZE flag, depending on the
  circumstances.

  Make  sure  that  you  call  FreePic() on the Picture once you have
  finished with 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_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.

PIC_Type
The data type of this picture, PLANAR, INTERLEAVED/ILBM 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.

VIDEOMEM - Allocates video memory that is displayable on screen.

RESIZEX - Resizes the picture so that it fits the given PIC_Width. If this flag is not set then the picture will be clipped.

RESIZEY - Resizes the picture so that it fits the given PIC_Height. If this flag is not set then the picture will be clipped.

RESIZE - Sets the RESIZEX and RESIZEY flags.

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

RESULT Picture - Pointer to a picture structure, NULL if error.

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

1.32 Master.GPI/UnpackPic

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

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

  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
```

1.33 Master.GPI/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.

  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
```

1.34 Master.GPI/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
  GMS).  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  following  flags  when making your memory
  allocation:

  MEM_PUBLIC if other programs will be accessing your memory.

  MEM_UNTRACKED  if  you  don't  want  resource  tracking  on  your
  allocation.  You should only use this flag if the memory is part of
  a complex resource that is already being tracked.

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
        -12(MemBlock)  will contain  the size of your allocated
        memory.   You  can  read this value, DON'T write to it.
        You  can  also  check  for  valid memory allocations by
        looking   at   the   ID   header.   "MEMH"  is  placed  at
        -4(MemBlock),  and  "MEMT"  is placed at the end of the
        memory block.

SEE ALSO
  FreeMemBlock
```

1.35 Master.GPI/FreeMemBlock

```
NAME  FreeMemBlock -- Free a previously allocated mem block.

SYNOPSIS
  FreeMemBlock(MemBlock)
               d0

  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 send a message to IceBreaker
  and leave the memory block in the system.
```

```
  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
```

1.36 Master.GPI/WriteDec

```
NAME  WriteDec -- Outputs a Number as decimal formatted text.

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

  APTR WriteDec(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.
```

1.37 Master.GPI/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().
  Note  that  this function does not free any individual objects that
  have been initialised - it just frees the file.

  This function will ignore null and invalid object bases.

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

SEE ALSO
  LoadObjectFile
```

1.38 Master.GPI/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)

  If  the  object is an identifiable tag list (eg TAGS_BOB) then this
  function  will  preprocess  it  into  its  native  structure.  This
  involves  some  memory  allocation,  so you must free the structure
  using   the   relevant   function   later   on.   For  example, calling
  GetObject()  on a TAGS_BOB object would return a Bob structure that
  at some point must be freed by FreeBob() or FreeStructure().

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
```

1.39 Master.GPI/GetObjectList

```
NAME  GetObjectList -- Get more than one object from an object file.

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  OBJECTLIST,0
    dc.l  <Name>,<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 <Object> field points to
         the relevant object that was found.

SEE ALSO
  GetObject
```

1.40 Master.GPI/CopyObject

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

SYNOPSIS

FUNCTION

INPUTS

RESULT

SEE ALSO
```

1.41 Master.GPI/FindGMSTask

```
NAME  FindGMSTask -- Find the GMSTask structure for the current task.
```

```
SYNOPSIS
  GMSTask = FindGMSTask()

  struct GMSTask * FindGMSTask(void)

FUNCTION
  This  function  will return the GMSTask structure for the task that
  called  it.  The GMSTask structure is used for storing data that is
  specific  to  your  task  -  things  like  preference  settings for
  example.   Almost all of the GMSTask fields are private and you you
  cannot write to this structure unless you are a GPI.

  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 structure.

SEE ALSO
  games/tasks.i
```

1.42 Master.GPI/CloseGMS

```
NAME  CloseGMS -- Closes the Games Master System.

SYNOPSIS
  CloseGMS()

  void CloseGMS(void)

FUNCTION
  Before  your  program  exits you will have to call this function so
  that  the  system knows you are shutting down.  If you do not close
  GMS  before  you  exit  you  will  leave certain memory allocations
  unfreed and there may be other adverse system effects.

  This  function  will  perform  a resource tracking check, so if you
  have  not  freed any system resources you will be notified here (if
  you  get a yellow alert box, you will need to use IceBreaker to get
  a detailed list of the errors).

  This  function is used in the StartGMS macro and the gms.o file, so
  C  and  assembler programmers should not need to call this function
  explicitly.

NOTE  Remember  that  you  may  not  call any GMS functions after calling
  CloseGMS().
```

1.43 Master.GPI/DebugMessage

```
NAME  DebugMessage -- Send a message to the debugger.

SYNOPSIS
  DebugMessage(Type, String)
          d7     a5

  void DebugMessage(ULONG Type, char *String);

FUNCTION
  Sends  a  message  to  the  GMS  debugger  if it is active.  If the
  debugger is not active then this function does nothing.

  This  function  is  intended for use in the GPI's, but you may also
  use  it  in  standard programs to send your own debug messages.  If
  this is the case then you will want to use the DBG_Message type and
  supply a String pointer.  The message will show up in bold text, so
  you can easily identify it in the debug output.

  GPI PROGRAMMERS
  You will find that there is a debug type for all GMS initialisation
  functions.  If you are going to use DebugMessage() inside your GPI
  it  must  be  called at the start of each function, as the debugger
  will  want  to  pick up your parameters.  You may also use the STEP
  flag  in  the  Type  parameter  so  that  the  debugger can provide
  tree-formatted  output.   If this is the case you will also need to
  call StepBack() at the end of your function.

INPUTS  Type   - One of the debug codes as described in games/debug.i.
  String - Optional   string   used  by  some  debug  codes  such  as
     DBG_Message.

SEE ALSO
  ErrorMessage
```

1.44 Master.GPI/ErrorMessage

```
NAME  ErrorMessage -- Send an error message to the debugger.

SYNOPSIS
  ErrorMessage(ErrorCode)
      d0

  void ErrorMessage(ULONG ErrorCode)

FUNCTION
  Sends an error message to the GMS debugger if it is active.  If the
  debugger is not active then this function does nothing.

  This  function  is  intended for use in the GPI's, but you may also
  use  it  in standard programs to send your own error messages.  The
  error  will show up in bold text so that you can easily identify it
  in the debug output.

INPUTS  ErrorCode - Standard GMS error code as described in games/games.i.
```

```
SEE ALSO
  DebugMessage
```

1.45 Master.GPI/AddTrack

```
NAME  AddTrack -- Add a resource tracking node.

SYNOPSIS
  Key = AddTrack(Resource, Data, Routine)
  d0      d0    d3   a0

  ULONG AddTrack(ULONG Resource, ULONG Data, void *Routine);

FUNCTION
  This  function  is  intended for use by GPI's but you may use it in
  your program if necessary.

  About Resource Tracking
  Each  GMS  task has a special list of all system resources that are
  currently  in  use.  Each  resource  has  its  own  list node that
  uniquely  identifiess  it  (eg  each  memory allocation has its own
  resource node).  When a particular resource has been freed the list
  is checked and its related node is deleted.  When the program shuts
  down,  any resources that are still in the list can be freed by GMS
  so that they are not lingering in the system.

  It  is  important to acknowledge here that resources are preferably
  tracked  in  their  most  basic  form  rather than the most complex
  types.  For  example  a  call  to  InitBob()  results  in a lot of
  calculations  and data storage, but when we look at it in detail we
  find  that  it  consists  entirely  of  memory.  Therefore only the
  memory needs to be tracked and a special track on the bob itself is
  not  necessary.  This  keeps  resource  tracking  very  simple and
  efficient.

  When you call AddTrack() you need to give it a resource ID which is
  made  up  from  one  of  the  following  resource  types.  This is
  important  as resources are freed from the system in an order based
  on these ID's.  The order looks like this:

    1. Free all hardware based resources (blitter, sound, etc).
    2. Free complex resources (both hardware and software).
    3. Free customised resources (user defined types etc).
    4. Free memory (always free memory last).

  As  you  can  see  the  correct order is vital, if memory was freed
  first  it  would  have adverse affects when freeing the complex and
  user resource types.  Always make sure that you give the correct ID
  when  describing  your  resource.  The  ID's  are  RES_MEMORY,
  RES_HARDWARE,  RES_COMPLEX  and  RES_CUSTOM,  as  outlined  in
  games/tasks.i.

  Passing  a  Data  pointer  is  optional,  but  you  will need it to
  uniquely  identify  your resource later on (eg AllocMemBlock() uses
```

```
  this to store a pointer to the allocated memory).

  The  Routine  points  to  code that will free the resource if it is
  still  in  use  when  the  program  exits  (or if SelfDestruct() is
  called).   It will be passed the Data field of the resource node in
  register  d0,  and the Key field in register d1.  This routine must
  save  all  the  CPU  registers  that  it uses and can only free its
  resource  -  it  may  not make new allocations of any sort.  It may
  send debug and error messages, which will help working out what the
  problem is if anything goes wrong.

INPUT Resource - Correct resource identifier from games/tasks.i.
  Data     - Optional data pointer to store in the resource node.
  Routine  - Pointer to the routine to call when freeing the resource.

RESULT  Key - A  key  that  AddTrack()  has  used  to uniquely identify the
        resource.   You will need to store the key somewhere for when
        you call DeleteTrack().

SEE ALSO
  DeleteTrack, games/tasks.i
```

1.46 Master.GPI/DeleteTrack

```
NAME  DeleteTrack -- Delete a resource tracking node.

SYNOPSIS
  DeleteTrack(Key)
        d1

  void DeleteTrack(ULONG Key);

FUNCTION
  Deletes  a  resource  node  allocated from AddTrack().  If resource
  nodes  are not deleted they will stay linked to the program and GMS
  will attempt to free them when the program shuts down.

  Note that this function only deletes the resource node, it will not
  attempt  to  deallocate  the  resource  by calling its deallocation
  function.

INPUT Key - A key that was obtained from AddTrack().

SEE ALSO
  AddTrack
```

1.47 Master.GPI/InitDestruct

```
NAME  InitDestruct -- Initialise the task for use of SelfDestruct().

SYNOPSIS
  InitDestruct(DestructCode, DestructStack)
```

```
  void InitDestruct(APTR DestructCode, APTR DestructStack)

FUNCTION
  This is a special function that is called in the STARTGMS macro and
  gms.o startup file.  You should never call this function explicitly
  unless you are writing your own startup code.

  InitDestruct()  will  prepare your task so that it may be destroyed
  by  the  SelfDestruct()  function.   DestructCode must point to the
  exit code for your task.  The exit code must call CloseGMS() if you
  are  to free your tasks resources.  DestructStack must point to the
  correct  stack  area for your exit code, otherwise your task cannot
  return to the system correctly.

  To  see  an example of how this function works look at the STARTGMS
  macro in games/games.i.

INPUTS
  DestructCode  - Points to the code at which your task makes its exit.
  DestructStack - Points to the stack that will be used for the exit.

SEE ALSO
  SelfDestruct
```

1.48 Master.GPI/SelfDestruct

```
NAME  SelfDestruct() -- Destroys the task and frees resources.

SYNOPSIS
  SelfDestruct()

  void SelfDestruct(void)

FUNCTION
  Destroys  the  task  that called this function and then proceeds to
  free all of its resources according to the resource nodes.  This is
  a  completely  safe and effective way of destroying a task, and can
  be   used   for   deconstructing  a  task  when  it  has  got  into
  unrecoverable circumstances.

  You  must  have called InitDestruct() before calling this function.
  If  you  are  programming  in C or assembler this initialisation is
  already  in the STARTGMS macro and gms.o startup file, so this does
  not concern you.

NOTE  This  function  will not return.  However if InitDestruct() has not
  been  called  then the function will not be able to do anything and
  will return back to the task.

SEE ALSO
  InitDestruct
```

1.49 Master.GPI/GetPicture

```
NAME  GetPicture -- Gets the latest version of the picture structure.

SYNOPSIS
  Picture = GetPicture()
    d0

  struct Picture * GetPicture(void);

FUNCTION
  Allocates the latest version of a GMS picture structure and returns
  it back to you.  The structure fields will be empty so that you can
  fill them out to suit your requirements.  Before your program exits
  you  will need to free the structure, this is automatically done in
  the FreePic() function.

  You  have  to use this function if you do not want to use tag lists
  to initialise your pictures (remember that it is illegal to compile
  and use pre-initialised structures in GMS programs).

RESULT  Picture - Points to the latest version of a GMS picture structure
      or NULL if failure.

SEE ALSO
  GetStructure
```

1.50 Master.GPI/GetStructure

```
NAME  GetStructure -- Gets the latest version of a specified structure.

SYNOPSIS
  Structure = GetStructure(ID)
      d0       d0

  APTR GetStructure(ULONG ID);

FUNCTION
  This  function will get the latest version of any structure in GMS,
  so long as it has an ID and a related Get*() function (GameScreens,
  Pictures, Sounds and others fit this requirement).

  You  can  also  pass  it TAGS_* identifiers, eg TAGS_GAMESCREEN, as
  they can be uniquely identified.

INPUT ID - One of the ID's as specified in the games/games.i file

RESULT  Structure - The  latest  version of the specified structure or NULL
        if  failure  (caused  by lack of memory or unrecognised
        ID).

SEE ALSO
  InitTags
```

1.51 Master.GPI/InitTags

```
NAME  InitTags -- Initialise a structure according to a tag list.

SYNOPSIS
  ErrorCode = InitTags(Structure, TagList)
     d0     a0    a1

  ULONG InitTags(APTR Structure, APTR TagList);

FUNCTION
  This function is intended for GPI's but may be used by GMS programs
  if required.

  It  will  process a standard tag list and store specified values in
  the  given  structure, which should be empty although this is not a
  pre-requisite.   It is important that the tags themselves have been
  correctly  defined  using  the TBYTE, TWORD and TLONG flags.  Check
  the games/games.i file for examples.

  This  function  has  some software based memory protection and will
  prevent values from being written outside of the structure's memory
  area.  Detected errors will be sent to the GMS debugger.

INPUTS  Structure - Pointer to allocated structure memory.
  TagList   - Pointer to a standard GMS tag list (see tags).

RESULT  ErrorCode - ERR_OK if successful.

SEE ALSO
  GetStructure
```

1.52 Master.GPI/CloseFile

```
NAME  CloseFile -- Close an open file.

SYNOPSIS
  CloseFile(File)

  void CloseFile(struct File *);

FUNCTION
  Closes  a file that was opened by OpenFile().  Files must always be
  closed  after  use,  especially  if  they have been locked with the
  FL_LOCK  flag.   Unclosed  files  will  be detected by the resource
  tracking routine and sent to this function.

INPUT File   - Pointer to a file structure obtained from OpenFile().

SEE ALSO
  OpenFile
```

1.53 Master.GPI/OpenFile

```
NAME  OpenFile -- Open a file for IO operations.

SYNOPSIS
  File = OpenFile(FileName, Flags)
   d0      a0     d0

  struct File * OpenFile(char *FileName, ULONG Flags);

FUNCTION
  Opens a file ready for reading and writing.  OpenFile() is a little
  more  advanced  than  normal  file  opening  functions and features
  auto-unpacking  and  file finding capabilities.  Here are the flags
  that affect the functioning of OpenFile().

  FL_READ
  This flag is the default and will open the file for reading data.

  FL_WRITE
  Prepares  the  file  for writing data, starting at byte position 0.
  If  you  want  to  start writing from the end of the file, copy the
  FL_Size value to FL_Position after OpenFile() returns successfully.

  FL_LOCK
  Setting  this  will  lock  the file for exclusive access - no other
  process  will  be able to open the file while you are using it.  If
  this  flag is not set then the file will be open for shared access.
  Attempting  to  lock a file that is already open with shared access
  results in failure.

  FL_FIND
  This  flag allows OpenFile to use some intelligence and try to find
  the  file  if  it is not immediately located at the given FileName.
  The process involves a simple but powerful tree search.

  Example
  You  try  to  load  a file at Game:Data/PIC.Crocodile.  However the
  Game:   assignment   does   not   exist.   You  are being run from the
  directory  HD1:Game/Data/  which  is a directory above the required
  logical   assignment.   OpenFile()  will  find the file by using the
  following procedure:

  Open -  Game:Data/PIC.Crocodile  (FAIL)
    Data/PIC.Crocodile (FAIL - HD1:Game/Data/Data/PIC.Crocodile)
    PIC.Crocodile      (SUCCESS - HD1:Game/Data/PIC.Crocodile)

  Note
  The  file  finding  feature  is  limited to localised searching, no
  attempt  will  be made to do a tree search of all directories in an
  assignment (a lengthy process).

  FL_UNPACK/FL_PACK
  You  can allow the file to be automatically decompressed by setting
  this  flag.   For the convenience of the user you should always set
  this  flag  unless you have a good reason not to, eg if the file is
```

```
  quite  large  then  access  can  be  slowed  down,  or  if you have
  compressed the file with your own private format.

  FL_BUFFER
  Setting  this  flag  allows  the  file to be put in the file cache,
  which  will  speed  up  read/write  operations  significantly.  Its
  effect  is largely dependent on the user, as there are many options
  in  controlling  the  file  cache, such as size limits, how large a
  file can get before it is no longer eligible for caching etc.

  Do  not  set this flag if you are only using this file once, eg for
  loading in data in the initialisation section of your program.

  FL_SMART
  This  flag  is  a  convenient  way  of  setting  both  FL_FIND  and
  FL_UNPACK/FL_PACK.


INPUTS
  FileName - Pointer  to  a filename that locates the file on a given
       medium.
  Flags    - One or more flags as outlined above.

RESULT  File - Pointer to an initialised file structure or NULL if failure.

SEE ALSO
  CloseFile
```

1.54 Master.GPI/ReadFile

```
NAME  ReadFile -- Reads a given amount of bytes from a file.

SYNOPSIS
  ErrorCode = ReadFile(File, Buffer, Length)
     d0         a0     a1      d0

  ULONG ReadFile(struct File *, APTR Buffer, ULONG Length);

FUNCTION
  This  function  will  read  the  amount  of  bytes as determined by
  'Length', from the File and into the given memory buffer.  The read
  will  start at the position determined by the File->Position field.
  This field will be incremented to File->Position+Length if the call
  succeeds.

  If the Length exceeds the total size of the file then this function
  will  only  read  as many bytes as their are left in the file.  You
  can  always compare the File->Position and File->Size fields to see
  how many bytes are left to read in the file.

  The  file data will be placed in the cache for faster reading later
  on, if the file was opened with the FL_BUFFER flag.

INPUTS  File   - Pointer to a file structure obtained from OpenFile().
  Buffer - Pointer to a memory area in which the data will be written
```

```
     to.
  Length - Amount of bytes to read from the file.

RESULT  ErrorCode = ERR_OK if successful.

SEE ALSO
  OpenFile
```

1.55 Master.GPI/WriteFile

```
NAME  WriteFile -- Writes a given amount of bytes to a file.

SYNOPSIS
  ErrorCode = WriteFile(File, Buffer, Length)
     d0                 a0     a1      d0

  ULONG WriteFile(struct File *, APTR Buffer, ULONG Length);

FUNCTION
  This  function  will  write  the  amount  of bytes as determined by
  'Length',  from  the  memory  buffer  and into the given File.  The
  write  will  start at the position determined by the File->Position
  field.   This field will be incremented to File->Position+Length if
  the call succeeds.

  If  the  File->Position+Length  exceeds the total size of the file,
  then this function will increase the file size to cope with writing
  out the rest of the buffer.

  If  the  file was opened with the FL_BUFFER flag then the file data
  will be placed in the cache, for faster reading later on.

INPUTS  File   - Pointer to a file structure obtained from OpenFile().
  Buffer - Pointer to data that will be written to the file.
  Length - Amount of bytse to write.

RESULT  ErrorCode = ERR_OK if successful.

SEE ALSO
  OpenFile
```

1.56 Master.GPI/LoadObjectFile

```
NAME  LoadObjectFile -- Load an external object file.

SYNOPSIS
  ObjectBase = LoadObjectFile(Filename)
      d0

  struct ObjectBase * ObjectBase LoadObjectFile(char *Filename);

FUNCTION
```

```
  Loads  in a GMS object file, checks it for validity and returns the
  resulting ObjectBase back to you.

  Once  you  have  called  this  function  you  can  use  the  object
  processing  functions such as GetObject() to obtain your data.  See
  the  information on Object Lists for further understanding of these
  files.

  You will need to free the object file ater using it.

INPUT Filename - Pointer  to  the  file in which the objects are located.
       Note  that  the  file will be smart loaded, so unpacking
       and file finding applies.

RESULT  ObjectBase - Points to a private structure, NULL if failure.

SEE ALSO
  FreeObjectFile
```

1.57 Master.GPI/StepBack

```
NAME  StepBack -- Steps back the debugging tree in IceBreaker.

SYNOPSIS
  StepBack()

  void StepBack(void);

FUNCTION
  This function is intended for use by GPI's.  You may use it in your
  own  programs  if  you want to use IceBreaker's tree feature to aid
  your output.

  StepBack()  has to be used in conjunction with the STEP flag in the
  DebugMessage()  function.   See DebugMessage() for more information
  on  this.  Any time DebugMessage() is called with the STEP flag you
  will  need  to  call  StepBack() as this is the only way to get the
  tree  back to the position before you altered it.  If you forget to
  call  StepBack()  then  the  debug  tree will be permanently out of
  position.

SEE ALSO
  DebugMessage
```

1.58 Master.GPI/LoadPicFile

```
NAME  LoadPicFile -- Load in picture file.

SYNOPSIS
  Picture = LoadPicFile(Filename, Flags)
    d0       a1   d0
```

```
  struct Picture * LoadPicFile(char *Filename, ULONG Flags);

FUNCTION
  This  function  is provided as a quick and simple way of loading in
  any  picture,  without the need of structures or tags.  It converts
  the  Filename  and Flags arguments to a tag list which it passes on
  to LoadPic().  You will be returned a Picture structure if the call
  is successful.

  At  some  point  you  will  have to call FreePic(), which frees the
  allocated   structure,  the  loaded  picture  data  and  any  extra
  allocations such as the palette.

INPUT Filename - Pointer to the file (picture) to load.
  Flags    - Flags valid for use in the PIC_Options field.

RESULT  Picture - Points  to  a newly allocated Picture structure.  NULL if
      unsuccessful.

SEE ALSO
  LoadPic
```

1.59 Master.GPI/GMSForbid

```
NAME  GMSForbid -- Stop other tasks/processes from executing.

SYNOPSIS
  GMSForbid()

  void GMSForbid()

FUNCTION
  Stops  all  other tasks and processes from executing until you call
  GMSPermit().  This call will not turn off interrupts.

NOTE  This function has no effect in systems that do not multi-task.

SEE ALSO
  GMSPermit
```

1.60 Master.GPI/GMSPermit

```
NAME  GMSPermit -- Allow other tasks to continue their processing.

SYNOPSIS
  GMSPermit()

  void GMSPermit(void);

FUNCTION
  Reverses  a  previous  call  to  GMSForbid(), so that all tasks can
  continue their normal processes.
```

```
SEE ALSO
  GMSForbid
```

1.61 Master.GPI/

```
NAME

SYNOPSIS

FUNCTION

INPUT

RESULT

SEE ALSO
```