General

Paul Manias

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

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

General

1.1 General Information

```
SYSTEM DOCUMENTATION
Name:      GENERAL INFORMATION
Version:   0.9 Beta.
Date:      October 1997
Author:    Paul Manias
Copyright: DreamWorld Productions, 1996-1997.  All rights reserved.

DESCRIPTION
This  documentation  covers how the Games Master System behaves, how it was
designed  and  why  those  design  methods  are  in place.  If you have any
questions in these areas hopefully they can be answered here.

           1. Data Checking
           2. File-Based Objects
           3. Error Codes
           4. Lists
           5. Structures
           6. Tags

CHANGES VERSION 0.9B
Added:  ERR_MEMORY
        ERR_NOSUPPORT

Edited: Tags Information (example code)
        List Information (LIST2)
        Almost all of this document has been edited
         in various parts.
```

1.2 File-Based Objects

```
                            FILE-BASED 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

Object data is compiled into standard Amiga segmented files. The easiest way to learn how it works is to view one; here is an example of a Screen and a Picture located in an object file:

```
  INCDIR  "GMSDev:Includes/"
  INCLUDE "games/dpkernel.i"

  SECTION "Start",DATA

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

Objects:
  dc.l  "OBJF"          ;File identification.

  ;The Screen 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_Screen:
  dc.l  "TAGS"          ;Object type.
  dc.l  OBJ_Picture     ;Pointer to next object.
  dc.b  "Screen",0      ;Name.
  even
.data dc.l  TAGS_SCREEN,0
  dc.l  GSA_AmtColours,16
  dc.l  GSA_ScrWidth,640
  dc.l  GSA_ScrHeight,256
```

```
  dc.l  GSA_Attrib,CENTRE
  dc.l  GSA_ScrMode,HIRES|LACED
  dc.l  TAGEND

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

OBJ_Picture:
  dc.l  "TAGS"           ;Object type.
  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_ScrMode,HIRES|LACED
  dc.l  PCA_Options,IMG_RESIZE
  dc.l  PCA_File,.file
  dc.l  TAGEND

.file FILENAME "GMS:demos/data/IFF.Pic640x256"

  ;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 object-file, then call the GetObject() function. You need to supply the name of the object you wish to grab, the function will do the rest.

Note that identifiable tag structures (eg TAGS_SCREEN) will be pre-processed by GetObject(), so you will usually be returned a Screen object that already contains the values from the tag-list. Your next step would then be to write out your own fields, and then call Init() to complete the initialisation process.

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 to NULL before calling the function. If you place something in the <Object> field then GetObjectList() will ignore that particular entry. You may also mix different kinds of objects in the same list, eg Screens and Sounds if that makes things easier for you.

1.3 Structure Layout

STRUCTURE LAYOUT

All structures have been designed with just one commonality: They all start with a standard system header. Following this are whatever fields are relevant for that structure type.

The structure header looks like this:

```
    STRUCTURE Head,0
  WORD    HEAD_ID
  WORD    HEAD_Version
  APTR    HEAD_SysObject
  APTR    HEAD_Stats
  LABEL   HEAD_SIZEOF
```

The ID consists of a word length object identifier. An example for Screens is ID_SCREEN. The ID is important to uniquely identify what class the object belongs to. Whenever we want to identify an object as quickly as possible, this is where we look first. Obviously it is important that this ID is correct at all times, otherwise you will confuse the system.

ID's can be used for more than just examining structures. One such is example is the LIST ID header, which tells a function that it needs to perform the same action to two or more structures (you can read more about this in Lists).

The Version field can be used for jump tables to deal with the various structure types and handling the future expansion of the structure. It starts at 1 and goes upwards.

The Stats field is reserved for low-level system use and cannot be read by normal programs.

The SysObject field points to the object's class details and should not be read unless you really know what you are doing. It serves no purpose to 99% of developers but is very important to the system, so try to leave it alone.

STRUCTURE AUTO-INITIALISATION

A standard policy for initialisation is to initialise all empty fields to either the user defaults, or values determined by any related fields. For

example, omitting the Width and Height values from a screen would cause the screen to open at the user's Width and Height defaults. On the other hand if you were to omit the child bitmap's Width and Height settings, these would inherit the values present in the parent screen's Width and Height.

Sometimes if there is a file present, the fields will receive values from that file's header structure. For example, IFF pictures will fill out most of a picture object when one is loaded.

The only fields that are not auto-initialised are the ones containing programmer flags (Attrib, Option) and Data fields.

FUTURE COMPATIBILITY

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 via a call to Get() or Init(). This rule solves all future concerns in relation to structure handling and size increases.

1.4 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 objects without having to make heaps of function calls. Lets say you wanted to load in 10 sounds from your hard-drive using Init(). Normally Init() takes a standard object, 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  LIST1               ;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  DPKBase(pc),a6
  lea     SoundList(pc),a0    ;a0 = Pointer to the soundlist.
  sub.l   a1,a1
```

```
  CALL    Init
  tst.l   d0
  bne.s   .error

Or: if (Init(&SoundList,NULL) IS NULL) {
     return(ERR_FAILED);
      }
```

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.

There is a second LIST type, suitably referred to as LIST2. This is a special list intended for the initialisation and freeing processes, and looks like this:

```
List: dc.l  LIST2,NULL
  dc.l  <TagList>,<Object>
  dc.l  LISTEND
```

The advantage of this list is that you can specify tags on the left, then when you call Init() it will place pointers to the allocated objects on the right. This makes things a lot easier when you want to Free() all of your object pointers.

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

1.5 Tags

TAGS AND TAGLISTS

Tags are supported in a way that is essentially identical to the Amiga OS. The only major difference is that an internal change in design allows them to be processed much faster.

Tags allow you to support all future structure versions, and they are very convenient for use in C. Because pre-compiled structures are illegal, you will have to use tags a lot. Make sure that you look at the demos so that you understand how to use them. The most important function with regards to tags is Init(), or for C programmers InitTags(). There are not many other functions that require such heavy use of tag lists.

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_SCREEN,NULL
  dc.l  GSA_Width,320
  dc.l  GSA_Height,256
  dc.l  TAGEND
```

If you do not give a structure pointer (as in this example), the structure will be allocated for you, via a call to Get(). The newly allocated structure will be placed in the structure pointer in your tag-list (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 system debugger like IceBreaker.

Here is an example of using tags in C:

```
      struct Screen *Screen;

      if (Screen = InitTags(NULL,
           TAGS_SCREEN,   NULL,
           GSA_Palette,   Palette,
           GSA_ScrMode,   LORES,
           GSA_Width,     320,
           GSA_Height,    256,
           GSA_ScrAttrib, DBLBUFFER,
           TAGEND)) {

         /* Code Here */

      Free(Screen);
      }
```

STEPPING INTO CHILD OBJECTS

A very interesting and quite important feature of the tags support, is the fact that you can "step into" child objects. For example the CardSet object has its own relevant fields, but as it is based on the Bob object, it also inherits the use of the Bob fields and its functionality. Now if you are initialising your CardSet object, you might want to set one of the Bob fields, such as the X and Y coordinates. But how do you do this if the fields are in an underlying structure? Here's how:

```
      struct CardSet *CardSet;

      if (CardSet = InitTags(NULL,
           TAGS_CARDSET,  NULL,
           CSA_Source,    file_CardSet,
             CSA_BobTags,   NULL,      <- Gain access to the Bob object.
             BBA_XCoord,    100,       <- Set the Bob X Coordinate.
             BBA_YCoord,    125,       <- Set the Bob Y Coordinate.
             TAGEND,        NULL,      <- Return back to the CardSet tags.
           TAGEND)) {

         /* Code Here */

      Free(CardSet);
      }
```

Notice how the prefix for the tags changed from CSA_ to BBA_ once we

entered the Bob object, and that we ended the Bob's section with TAGEND and NULL. We also indented the list to make it very clear that the tag list was moving into a child object.

SPECIAL FLAGS
Lastly there are also some special flags that you can use for advanced Tag handling. It is unlikely that you will ever need to use these, but they are available if you require them. 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 Error Codes

ERROR CODES

A universal set of error codes is used by all 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 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_TYPE
Bitmap Type not recognised or supported, eg currently True Colour modes are not available.

[10] ERR_MODULE
This function tried to initialise a module 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
Raster object was missing from Screen->Raster.

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

[15] ERR_FILEMISSING
File not found, this occurs when a FileName references a file that does not exist.

[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.

[19] ERR_ARGS
You tried to pass invalid arguments to this function, such as a NULL value where you should have placed a pointer.

[20] ERR_NODATA
This function expected some data which you have not supplied.

[21] ERR_READ
Occurs if you attempt to read a file and a failure occurs. Perhaps there is nothing left to read, the file object has been invalidated, or there are surface errors on the media in question.

[22] ERR_WRITE
Occurs if you attempt to write to a file and a failure occurs. Usually this is due to a hardware problem such as lack of space or surface errors.

[23] ERR_LOCK
This error can occur if you attempt to get a lock on an object that is already locked out by another task, or if you try to lock an object that does not exist.

[24] ERR_EXAMINE
This error applies to file and directories. It can occur for various reasons, a probable cause could be a corrupt/invalidated area of a disk structure.

[25] ERR_LOSTCLASS
This is a very serious error, which occurs if an object loses its reference to its SysObject. This basically means that the Object->Head.SysObject has been cleared or invalidated.

[26] ERR_NOACTION
If you call an action on an object, and if the object does not support that action, you will receive this message. An example of this could be:

```
Draw(Segment);
```

[27] ERR_NOSUPPORT
Can be caused if an object fails to initialise, often due to an unsupported data format.

[28] ERR_MEMORY
General memory error, such as no memory available, or memory too fragmented.

1.7 Data Checking

DATA CHECKING

AND

VALIDATION PROCEDURES

This system has been designed to allow for the inclusion of simple and efficient data checking mechanisms. All kernel based functions support null handling, data checking, and data validation. Due to the presence of IceBreaker you can quickly find out where a program has failed, as functions report errors directly to you in english (not error codes or software failure numbers).

GARBAGE PROTECTION

The simplest example of garbage protection is that of functions checking that they have been passed the correct data structures before they actually do anything with them. This offers the programmer more security when developing, as passing the wrong data to a function is not uncommon and can often result in disaster. In the past, protection from such actions has been employed at the level of a language compiler, such as C's type

checking feature. However this has obvious problems as there is never any assurance that what you are passing is correct - there is no way of knowing that the data has not been tampered with, if the pointer is simply pointing to the wrong structure or if the object was not initialised in the first place.

Therefore the only sure-fire way of protecting a system and the programmer from such accidents is at the function level, where the data is checked for validity before hand and not processed until it passes the various tests employed. This does not mean checking each field for valid values, a simple ID test can perform wonders. Surprisingly I do not know of any other OS that consistently offers such a feature.

Other garbage protection features include software based memory protection. This is significantly different to hardware based memory protection which requires the addition of a memory management system with the CPU. Although memory protection on a software level will always be less stringent than at hardware, it has an advantage of being able to deal with errors in a more suitable way, rather than exceptions being generated that often result in the program being shut down by the OS. A good example of all this is the TagInit() function.

TagInit() takes an empty structure and a list of data tags, then processes the data tags and writes out the values to the given structure. The procedure to do this is straight forward and takes no more than a few lines of assembler to implement. However if the tag list has bad data values there is nothing to stop data being written outside of the structures memory boundary. To stop this TagInit() finds out the size of the destination structure and compares it to the destination given in each tag. If a problem is discovered the function stops operations, passes an informative message on to IceBreaker, then returns to the program with an error code. The program can then exit in its own way, and the programmer can immediately find out what went wrong without "exception code X at address $X" and a list of incomprehensible assembler registers.

DATA INTERFERENCE

There are two types of data interference: 'Accidental' interference (which is the result of a programming error), and 'Deliberate' interference (you change the data of a program on purpose).

Deliberate Data Interference is a feature written into many GMS objects and functions. When a system routine finds a data value that is in error or is extremely inappropriate, it will be changed. This can prevent a program from crashing, or unexpectedly exiting for small mistakes. An example of this might be attempting to open a screen at 640x1024, when the user's system only supports 640x512. In this case the screen object will alter its height so that it opens at the correct screen size. To alert the programmer of the mistake, a message will be sent to the system debugger. In the current AmigaOS this checking never happens, and a common problem is with windows that open at sizes that do not fit on screen.

Here is a second example:

Lets say a programmer was loading a picture that was 4 planes in depth.

Although this was specified in the picture structure he forgot to set the screen type to ILBM. Because it was already set in his preference settings for GMSPrefs it works anyway, so the problem goes unnoticed. Later, a user with a setting of CHUNKY8 tries the program. Now normally this could crash the system or cause the program to never work at all, because a CHUNKY8 screen cannot have a plane setting of 4. However Init(Picture) will pick up on this and immediately change the setting to 8. The problem is now solved, and the program continues to work fine.

This kind of checking and comparisons are often used and it allows a certain amount of extra future-proofing to the general functionality. A small programming flaw made now, which might not show up until future GMS versions, can now be dealt with quickly and without incident.