Default

Paul Manias

COLLABORATORS

	TITLE : Default	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Paul Manias	October 30, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Default

1.1 The Games Master System V1.0

T H E G A M E S M A S T E R S Y S T E M

BY PAUL MANIAS

VERSION 1.0

GENERAL DOCUMENTATION

1.2 Introduction to the Games Master System

This introductory text is taken directly from the web pages ↩
:

http://gms.ethos.co.nz/gms/

WHAT IS THE GAMES MASTER SYSTEM?

The Games Master System is an Amiga software product that has been in development since April 1996. Essentially, GMS is an Operating System specifically tailored towards a gaming focus. It has all the standard features of an OS (file-system, graphics handling, kernel etc), plus some very neat features for game developers. This includes easy handling of joysticks, sound play, special effects, colour handling, pictures etc. It is fully object oriented, and you won't even need a special language to develop 100% OO applications. GMS supports C, Assembler, Amiga E, Basic and an Amos extension is currently in development.

The system is fully modular and is designed to be ported to other computer platforms. It can run on top of other operating systems or as an independent OS. A PC version is to be developed in late 1998 and will appear some time in 1999. For the first time, games developers will be able to port their products to other platforms without having to make any changes. Features like this could enhance the entire game industry, with developers being able to support large numbers of platforms with just one set of source code. Computer systems with small market share (Amiga, Mac, Acorn etc) could get back on equal ground with the PC once again. It is expected that applications written in C could be distributed in a garbled/secure state, and then compiled on a user's platform. This would give the nearest equivalent of Java style applications without the sacrifice in speed.

Open System

The most exciting part of all - GMS is a system that is very open to third party development. After the up and coming official release, we will start releasing much of the original source code into the general public so that the system can be enhanced by developers all over the world. This will also make it easy for hardware manufacturers to develop their own drivers for graphics, sound cards etc with a minimum of effort. We also envision the appearance of neat device drivers that can run games in windows rather than screens for example. The future possibilities are literally endless.

User Support

When this project first started, one of the first things that was considered was the user support. Far too often this is completely ignored or is not possible for game developers, often because they have dead-lines to meet and the necessary time is unavailable. Games players are becoming

more sophisticated and expect better support, especially for their hardware. Take the release of "Time Crisis" on the PSX as an example. The programmers developed an interface which worked with a particular kind of light gun - and as luck would have it, half of the existing light guns didn't work with it. GMS solves these issues by abstracting the interface from the programmer, and providing a standard interface on top of it. You could plug in a joystick and fool any program into thinking it's a mouse for example (can be useful if your mouse is broken :-).

The best feature for users though, is the GMS Preferences program.. This allows you to select levels of mode promotion, type of joystick used, C2P routines, task priorities, screen properties, and so on. This solves a lot of the moans and gripes that users have had in the past, and since this is all transparent to the programmer, user support is easily achieved. GMS is the only OS to provide such extensive support to games players, and knowing what you guys are currently putting up with, it will continue to be for some time.

Anything else?

Look around the rest of the site to get more detail on the things mentioned here, and other things that we haven't gone into yet. The GMS binaries and documents are available on Aminet, in dev/misc/gms_dev.lha and dev/misc/gms_user.lha. Remember to ask us if you have any questions about the project!

Return To Index

1.3 OverView

OVERVIEW OF THE GAMES MASTER SYSTEM

Project GMS started in the beginning of April 1996, in an effort to provide games support in the Amiga OS. The overall aim is to write the best games interface we possibly can, which should eventuate into a system that everyone can enjoy. Although the development of GMS is largely controlled by myself (Paul Manias) I would like people to see it as a project of the Amiga community and I am completely open to ideas and comments. The current objectives of the project are:

1. To erradicate the need to bash the hardware from within games.
2. To make it easier to migrate from the current Amigas to the new Power Amigas.
3. To make games programming easier, faster, and more productive.
4. To give users the ability to modify any program to suit their requirements.

GMS has been designed to be fully extendible in ways that will make future improvements very easy to implement. The system is split into a number of sub-sections: The kernel, the modules, the debugger, and the preferences

program. This is further enhanced by identifiable data objects, which allow us to write enhanced system objects in the future, without overhauling the functions. GMS has no problems with future compatibility, since hard-coded structure definitions are disallowed and tag-lists are very well supported.

Looking to the Future
In 1998 I will release all of the module source code to the public domain. The only thing that I will continue to develop is the Kernel, Preferences, and System Debugger. I will also continue to define all of the standards, include files, documentation and so on. In short, I will be moving into more of a system management role.

This means that somone else will have to write the modules, and that someone is you. With the source freely available, I am hoping that people will begin to develop support for graphics and sound cards, enhanced modules, fixing bugs and so on. I have ended up developing a project that is finally too big for me to handle alone, and I think this is the best way to get everyone involved. The PPC version is entirely dependent on you, because I will not be converting the modules.

Please help me all that you can, because I cannot do it without you.

Return To Index

1.4 Documentation Standards

DOCUMENTATION STANDARDS

Originally the autodocs for GMS were modelled on the autodoc format from the 1980's (which was a fairly good design back then) but the GMS docs are now written in a completely new format. This change was made because the arrangement of documentation for an object oriented OS needs to be considerably different from an OS that is not. With the introduction of AmigaGuide and HTML formatting, there have also been considerable advances in the way that developer documentation can be written.

Fortunately most of the formatting is quite straight-forward and the standards are easy to grasp hold of. Here you will find the documentation templates and information on what certain keywords mean.

Module Documenation
The documentation for each module starts with a header that describes the details of the document and the module it covers. The following is an example taken from the kernel documentation:

```
SYSTEM DOCUMENTATION
Name:       DPKERNEL
Version:    1.0
```

```
  Date:       March 1998
  Author:     Paul Manias
  Translator: Julian Boibessot
  Copyright:  DreamWorld Productions, 1996-1998.  All rights reserved.
```

The field descriptions are as follows:

```
  Name:       The name of the module.  This should contain a link to the
              function list for the module.
  Version:    Standard version numbers such as 1.0, 2.5B etc.
  Date:       The date that this documentation was written (not the date
              that the module was released).
  Author:     Who wrote the module/documentation.
  Translator: If this is a translated document, the name of the translator
              goes here.
  Copyright:  Use for standard copyright notices.
```

Following this header may be a list of changes, which describe all the alterations made since the last time the document was updated.

Function Documentation

Each module document must have a page that lists any objects that it contains, and a list of all public functions. Each function name must link directly to a detailed description of what that function does. If necessary you may categorise your function list into groups, but keep everything in alphabetical order.

The description of each function must follow a strict standard, here is an example:

```
---
  FUNCTION
  Name:     AllocMemBlock()
  Short:    Allocate a new memory block.
  Synopsis: APTR AllocMemBlock(LONG Size [d0], LONG MemType [d1]);

  DESCRIPTION
  ...

  NOTE
  ...

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

  RESULT
  Pointer to the start of your allocated memblock or NULL if failure.

  SEE ALSO
  Kernel: FreeMemBlock()
          GetMemSize()
          GetMemType()
---
```

The first header, "FUNCTION", may be replaced with the keyword "ACTION" if

the function is an action call. Following the header are just three compulsory descriptors:

```
Name:     The name of the function.  Remember the () sign.
Short:    A one line description of what the function does.
Synopsis: A function definition that both C and assembler programmers can
          understand.  Notice that registers are shown in [] brackets,
          and that pointers to objects are preceeded with a * sign.
```

The Description then follows, and can contain as much text as you feel necessary to explain what a particular function does. You may specify important Notes in their own section if you wish.

The Input section elaborates on the Synopsis by saying what each argument is for.

The Result is a 1 - 3 line description on what the function returns. If the function returns void, do not use this section.

"See Also" is a section that specifies anything that relates to the function. This could be pointers to other functions, include files or objects. Make sure that everything is linked so that the reader can quickly navigate to other documents from here.

Object Documenation

Each object in the system is documented individually, with a high degree of detail. The beginning of an object's documentation is almost identical to the module documentation, e.g.

```
OBJECT DOCUMENTATION
Name:      FILE
Version:   1.0
Date:      March 1998
Author:    Paul Manias
Copyright: DreamWorld Productions, 1996-1998.  All rights reserved.
```

Following this is the second page of the object documentation which will contain: An object header, description of the object's functionality, an action list, a function list, and a field list. The object header is formatted like this:

```
OBJECT
Name:    File
Version: 1
ID       ID_FILE
Module:  Files
Include: "files/files.h"
Type:    Complex
```

This is fairly explanatory, except for the Type setting. The Type describes the complexity of your object, which can either be Simple or Complex. Simple types have a low number of structure fields (no more than 10) and are geared towards performing straight-forward functions. Complex objects are larger, may have a number of child objects, and many functions in its support.

Field Documentation
It is important that each field of an object is documented individually. The documentation format for a field is as follows:

1. Header
2. Description
3. Notes
4. See Also

A typical header looks like this:

```
FIELD
Name:         Height
Type:         WORD
Inheritance:  SrcBitmap->Height
To Change:    SetBobDimensions()
Status:       Read/Init
```

The Inheritance describes where the field will get its value if it is specified as NULL on initialisation. The "To Change" specification is the function that must be called in order to write to the field (if applicable). The Status is formed from one or more of the following:

```
Read  - If the programmer is allowed to read the field.
Init  - If the field can be written on initialisation only.
Write - Write at any time (field is dynamic).
```

If you set the Write field, then there is no need to specify Init.

1.5 Questions and Answers

QUESTIONS AND ANSWERS

I often get mail from people asking me questions about what you can and can't do in GMS. Here I will answer some of these questions, and hopefully this way everyone can benefit in learning more about how GMS works. If you have any questions you can mail them to me and they may appear here.

GENERAL
Q. Will you support new machines such as those from PIOS and the upcoming OS's like pOS?

A. Buy the 680x0 version, then I can quit my job and put more effort into this kind of support. Failing that, when the source code is released I hope that developers will pool their efforts and write hardware and interface drivers for any machine that you want.

Q. Is it possible to free structures that have not been initialised? [The idea being that it makes it easier to write the initialisation code]

A. Yes, a standard feature of GMS is to recognise structures that have not been initialised. Many functions check if you have passed them null or

invalid structures, so the security in this area is very solid.

BLITTING

Q. If I want to have 5 Bobs with the same graphics, may I initialise the first with NULL in Bob->MaskData and GENMASK set, then copy the pointer created to the other structures and init them with GENMASK cleared? Otherwise the masks would be created again several times and waste memory.

A. Yes, it is legal to copy masks generated in one Bob over to another Bob. Just remember when you free the "master" bob, all Bob's containing pointers to its masks will become invalid. For this reason make sure that you free the master bob last.

Q. How does CPU assisted blitting work when parallel drawing means that there could be an instance of data overlap (CPU and blitter draw to the same area at the same point in time)?

A. It's a clever trick... What you do is start the blitter drawing the first 5 lines or so of the bob. While it does that you use the CPU to start drawing 5 lines from the bottom of the bob. When the CPU finishes with its section it checks on the blitter. If the blitter has finished then the CPU sets it blitting the next 5 lines and the CPU continues on. This keeps going until the blit is finished.

Q. I have successfully initialised a Bob with a Picture attachment. when I draw it the dimensions are correct but the graphic is corrupt.

A. Make sure that you have specified the MEM_VIDEO or MEM_BLIT flags in the MemType field of the Picture's Bitmap. If you forget to set one of these flags then the blitter will probably be attempting to blit from fast memory, resulting in corrupt graphics.

SCREENS

Q. How do I create a Screen that can scroll infinitely in all directions?

A. You have to create the Screen with a Bitmap that is double the size of the Screen in both width and height. By scrolling around this area and blitting tiles at the top, bottom, left and right sides of the screen, you can give the illusion of an infinite scroll.

Return To Index

1.6 How to write new modules.

WRITING NEW MODULES

Anyone can write a module, but there are a couple of things you have to do first. You must decide what kind of module you are going to write, understand object orientation under the DPKernel, look at the source to other modules and read the module guidelines. If you don't know about child classes, hidden objects, or system classes, go and read the SysObject document for details.

A module can do basically two things: It can contain system objects (eg the blitter module carries the Bob and MBob objects), and it can contain functions that perform certain actions (eg DrawPixel()). The system modules are pretty much focussed around objects, but you may want to write one that consists entirely of functions. On the other hand, the JoyPorts and Picture modules are entirely object orientated, leaving the kernel to support the functionality of their objects.

An example of writing a new module
Let's say you wanted to add jpeg support to the Picture class. Because you will be adding support to a class that already exists, you will be creating a child class object & module. The first thing to do in this case is write to paul@ethos.co.nz and ask if such a module has already been written. You may change your mind if it's already been done. If everything is well you will be sent back some information on what to call your module (in this case, probably "jpeg.mod") and if necessary, a unique module identifier (not required for child modules).

The next thing to do is write an object referencing file. You can view a list of reference files in System/References/. You'll notice that these files are extremely small, so obviously it won't take long to write one. Our particular reference file will end up looking like this:

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

  SECTION "Reference",DATA

Start:  dc.l  TAGS_REFERENCE,0
        dc.l  REFA_ObjectID,ID_HIDDEN
        dc.l  REFA_ObjectName,.name
        dc.l  REFA_ModName,.module
        dc.l  REFA_CheckFile,.checkfile
        dc.l  TAGEND

.name   dc.b  "Jpeg",0
        even

.module dc.b  "jpeg.mod",0
        even

.checkfile
  cmp.l   #$FFD8FFE0,(a1)
  bne.s   .chk0
  cmp.l   #"JFIF",6(a1)
  bne.s   .chk1
.chk2 move.l  #99,d0
  rts
```

```
.chk1 move.l  #60,d0
  rts

.chk0 move.l  #00,d0
  rts
```

[Notice that although ID_HIDDEN was specified as the ObjectID, this only concerns referencing. The actual Jpeg object itself will be initialised as a child object later, not a hidden object.]

At this point I'm going to stop, I'll come back and write some more later :-)

Conditions of Module Development

Modules are grouped into two different types: Class modules, which carry master classes and functions, and Support modules, which may carry functions, hidden objects and child classes. 80% of module programmers should fit into the second category, for which there are minimal requirements. Class modules are developed under fairly strict guidelines because they are much more important. Here are the conditions:

Class Modules

If you intend to write a module that will contain the code for a master class, you MUST:

1. Register the module and object(s) by writing to paul@ethos.co.nz. [You will receive the necessary ID's to start development].

2. Write accurate and comprehensive documentation for the object and module over their continued development.

3. Four weeks before you intend to release the first version, you must send the module and any relevant information to paul@ethos.co.nz to get final approval of object and function definitions. Alternatively you may show what you're doing on a regular basis to keep everything on track.

4. If in the event that you stop writing your module you should pass all of the development information (ie source code) to a person of your choice. Alternatively you can send it to DreamWorld Productions so that we can find someone that wants to continue its support. We don't like to see modules created and then dropped without continued development!

Failing to support the above guidelines will cause DreamWorld Productions to stop giving your work any recognition what so ever. We also reserve the right to prevent your module from running even if it has been installed on a user's system (everything stops at the kernel if we want it to). Remember there may be a lot of people using your module, so we must ensure that it's 100% OK and can be upgraded for the future.

Support Modules

If you are writing a module containing a set of functions, and/or 1 or more hidden objects or child classes, you should:

1. Register the name of the module by writing to paul@ethos.co.nz. [This is done simply to prevent naming conflicts in the System/ directory.]

Return To Index

1.7 Really cool features!

CURRENT FEATURES OF THE GAMES MASTER SYSTEM

This is just a summary of the major features that have so far been implemented. Not all new features and changes have been documented here. For the complete low-down on all features of GMS check the developer information files.

* Completely object oriented system design, covering all aspects of OO including data/function inheritance and polymorphism. This allows for much more powerful programming, data abstraction and modularity.

* Multiple platform capabilities. A GMS program compiled on a 680x0 Amiga could also be run on a 680x0 Atari or Mac, all you would need is the necesary drivers. All PPC compilations will have the same feature and can also be 680x0 compatible through emulation.

* Resource tracking is fully implemented, a task can exit and all its resources will be completely freed. A SelfDestruct() function allows a task to abort itself at any time and the system will free its resources - extremely useful for debugging purposes. A user may force a task to abort simply by pressing L-AMIGA and DELETE, 100% safe.

* Debugging support implemented into all initialisation functions, no need for unnecessary patches to get debug information or track system calls. A debugger utility exists for receiving and displaying this data in real time.

* Transparent Chunky-To-Planar, which means it won't bother wasting time with conversions or copying if chunky mode is already available in the hardware.

* Fast blitter functions for drawing bob's, copying for screen buffers, 3 different screen clears, auto background saving and clearing for bob's. Also includes Pixel and Line drawing functions, and support for list's for very fast mass-drawing operations. CPU assisted blitting means that drawing speed is no longer limited to blitter throughput.

* Sound support includes: Support for sound priorities, intelligent dynamic channel play-back, channel modulation for special effects, IFF support.

* Proportional colour fading, functions are: PaletteMorph(), ColourMorph(), PaletteToColour() and ColourToPalette(). Support for setting speed and colour ranges.

* Full support for raster/copperlists, with effects such as: ColourLists,

Mirror, Flood, and Palette Changes.

* Structure and object pre-processing, allows data to be changed separately from the main program. This makes GMS the first system to support up to 100% user editing of game data.

* Allows you to support all different kinds of input devices (joysticks, joypads, mouse etc) through just one simple function call. This enables you to support devices that don't even exist yet.

* User preferences program to allow full configuration of a game's functionality. This includes configuration for: Game/Task Priorities, Joystick emulation, Mode Promotion, Screen Settings, and more.

* Stable memory allocation and a freemem routine that will not crash your machine if you have written over your memory boundaries. Internal resource tracking ensures that GMS programs will not chew up your memory.

* Smart Saving and Loading of files, with automatic packing and depacking via XPK.

* 320k of assembler, E and C sources, demonstrating all uses of the library.

* All GMS programs can multi-task with no significant drop in speed or performance.

Return To Index

1.8 What language to use?

LANGUAGES

As GMS is no more than an extension to the OS, it can work with any language that you want it to. Currently supported languages in this archive are:

Assembler
C/C++
E

You could also use Blitz Basic, Pascal, Oberon and others if you know how, but I currently don't have any source demos or special include files to help you with those.

If you're new to programming then I would recommend starting out with C or E. In my opinion E is a little easier on the beginner, but C is more common place in other areas and you might find that useful. If you don't know which one to choose, try learning E first, and then C as they are

quite similar languages. Unfortunately E has some portability problems, so if you intend to do things properly you will definitely need to upgrade to C at a later date.

If you want to write really fast games, you will have to learn some assembler. With GMS learning assembler is quite easy, as you don't have to think about programming the hardware registers. Look at the demo sources and make up your own mind if you want to learn it or not. Using GMS you could become a fairly adequate assembly programmer in as little as 2 months if you have come from something like C. Bear in mind that assembler makes programs difficult/impossible to port to other CPU's.

WHAT COMPILER?

If you know what language you want to use, you will have to think about what compilers you should get. You can't program without a compiler! Here is my opinion on the most common and best compilers available:

Assemblers

The best assemblers are AsmOne, DevPac and PhxAss. I have all three of these and use each one of them for different situations. You don't need that many, but two of these programs are free, so it won't cost you anything.

AsmOne has an excellent source-level debugger and I recommend it to beginners, as you can observe how the 680x0 instructions work. I don't use it that much today, but it is useful and has some features that make it very easy to use. It also has the fastest compiler speeds that you could imagine. I got the latest AsmOne from the WWW, go to one of the Amiga Web searches and look for "AsmOne" to find it.

DevPac is a good, robust compiler with many options, but it's a little slow and hasn't been updated in a while. I recently moved to using PhxAss for these reasons. You can get DevPac from HiSoft and other software dealers, it's a commercial product so you will have to pay for it.

I have been using PhxAss for a while and have found it to be a very impressive asm compiler. It is compatible with DevPac sources and has very good compiling times. The package is regularly upgraded and it's freeware. Good work Frank Wille! You can get PhxAss from Aminet, just download it as dev/asm/Phx*.lha.

You will also need a text editor if you're using DevPac or PhxAss, I recommend CygnusEd as it's small and you can alter the TAB stops. This feature is important as it keeps your sources easier to write and manage. You will notice that all my assembler sources look strange unless you view them with CED or AsmOne. CygnusEd has been upgraded recently, so now is a good time to buy.

C Compilers

SAS C/C++ is what I use most often, it's very reliable and I've never had a problem with it. The documentation is very extensive, so you'll be able to get help for all your problems. This product is no longer officially supported, but you can order the remaining copies on the WWW. You may be able to get it from various software dealers. Dice C is a nice package but

it doesn't support any C++. It has recently been released as freeware, so it's worth getting as your first C compiler. You can get this one from Aminet, in dev/c/ I believe.

Again, you will need a good text editor for efficient programming. CygnusEd is ideal here, and I believe GoldEd is a popular choice as well.

E Compilers
There is only one E compiler available (EC) which you can get as part of the E package. You can get this from Aminet, along with everything else that you will need. You will probably have to register, although this program was put on a coverdisk some time ago.

Return To Index

1.9 Hints and Tips

GAMES MASTER SYSTEM

HINTS AND TIPS

This section is written to offer some friendly advice and tips on how to get full use from the Games Master System, and what tricks you can use to make sure your game runs at the highest speed possible. I'm still writing this section, but if you have a trick of your own that should be here, please write to me at paul@ethos.co.nz. Even though I wrote the system, I don't know everything that can be done with it :-)

1.1 GENERAL CODING TIPS

Less... equals More!

Never call the same routine twice in your main loop unless absolutely necessary. For example, look at this routine that calls Query() twice:

```
----
Loop: move.l  DPKBase(pc),a6
      move.l  KeyStruct(pc),a0
      CALL    Query
      move.l  KeyStruct(pc),a0
      move.l  KEY_Buffer(a0),a0
      cmp.b   #K_ESC,(a0)
      beq     Game_Over

      ...
      Rest of main loop
      ...

      move.l  KeyStruct(pc),a0
      CALL    Query
```

```
      move.l  KeyStruct(pc),a0
      move.l  KEY_Buffer(a0),a0
      cmp.b   #" ",(a0)
      beq     .Exit
      ...
      bra.s  Loop

KeyStruct:
      dc.l   0
```

Do this instead...

```
Loop: move.l  DPKBase(pc),a6
      move.l  KeyStruct(pc),a0
      CALL    Query
      move.l  KeyStruct(pc),a0
      move.l  KEY_Buffer(a0),a0
      cmp.b   #K_ESC,(a0)
      beq     Game_Over

      ...
      Rest of main loop
      ...

      move.l  KeyStruct(pc),a0
      move.l  KEY_Buffer(a0),a0
      cmp.b   #" ",(a0)
      beq     .Exit
      ...
      bra.s   Loop
```

As you can see the second version is faster because it doesn't make an extra call to Query(). Simple really, but it often happens to beginners and in large programs.

1.2 CONTINUATION OF TASK PROCESSING WHILE PAUSED

There are times when pausing of your main task (through WaitAVBL()) will be inconvenient if it is necessary to continually process information. On the other hand, if your program continues to run in the background it will steal the processor for as long as it continues drawing.

Lets say you are writing a game that can connect via the serial port for 2 player communications. If one machine was to stop its processing, the serial buffer will continue to receive information and could go into overflow, potentially causing you some problems when your task is reactivated. The easy solution to this is to activate a secondary task that will continue to process when the main task is paused. This is a simple procedure and only requires that you put all your communication handling into this separate task. Another method is to use an interrupt, although that is not necessary in this case.

An option that may be more convenient for the user in a TCP environment, would be to send out a message saying "This machine is temporarily paused" so that all other machines know that they must not send information to you. This will give any other TCP tasks running on the paused machine more time to send/receive data, eg for FTP.

1.3 SUPPORTING HIGHER RESOLUTIONS

Drawing high resolution graphics and supporting them as an option in your game is a worthwhile exercise, and will make the owners of more powerful computers happy. But it can be annoying to support - most developers make two copies of each picture file, one in lo-res and one in hi-res and then program the game to support both files. This can get in the way of programming the game itself and results in wasted time. In GMS there is a way to solve this problem.

Draw all the graphics in high resolution and use them as you normally would in your game. Use screen tag lists that accept the default screen dimensions from the user (do not set GSA_ScrWidth, GSA_ScrHeight or GSA_ScrMode).

Set the RESIZE flag when loading in the pictures and set PIC_Width and PIC_Height in accordance to the user's resolution in the screen that you opened.

Example: If the game graphics were drawn on a 640x512 screen and the user has asked for a LORES screen, then drop the picture dimension to 320x256 and load it in. The picture will be resized to fit the new dimensions and you now have the lo-res equivalent of your hi-res screen.

The next step is to proportionalise your bobs to the new settings. There are two fields to help you do this - PropWidth and PropHeight. These fields must contain the original dimensions of the Bob's source picture, which in the example above was 640x512.

Now, when you call Init(), the function will detect that the Width of the source Picture does not match the PropWidth setting (same thing for the height). It will then use a formula to alter your Bob's Width, Height and coordinates to reflect the new dimensions.

```
Bob->Width = (Picture->Width) / (Bob->PropWidth / Bob->Width)
```

That's it. There are some proportional demos in this archive, check those to see how easy the procedure is (note how there is no extra coding needed, just the addition of a few tags and the RESIZE attribute).

Return To Index

1.10 The Authors

THE AUTHORS

The Games Master System is written in Assembler and C by Paul Manias. Paul has 5 years 680x0 and games programming experience, and another 2 years in other languages like C and Pascal. Paul's favourite past-times are blowing his nose, staring at the ceiling, and lurking in basements. So far he has written two games of his own and contributed graphics to two other commercial ones. None of those games have beeen released (yet?), for all sorts of various reasons. Luckily this is not the case with GMS.

GMSPrefs was orginally written in E, by Richard Clark. Richard's favourite past-times are standing, sending morse code via blinking, and talking to suspicious items of furniture.

Many thanks to Graeme Chiu, who hosted the GMS pages from April 1996 - May 1997. To see our WWW pages, visit:

http://gms.ethos.co.nz/gms/

To see a detailed list of all the contributors to the GMS project, go to "The Authors" section on the WWW. Thanks to the people that send in useful bug reports, and to the many people that sent in ideas when the project first started (but we still need more!).

Return To Index

1.11 Resource Tracking

RESOURCE TRACKING

GMS is fully supportive of internal resource tracking, which means that it tracks resources without any effort from the programmer. Resource tracking is great for programming as it warns you if you have forgotten to free important system allocations when your program exits. This is not just memory, but also things like sound, blitter, video display, files and device allocations. This becomes a life saver in situations such as forgetting to free a hardware allocation like the blitter, as this would normally cause a system deadlock and you would have to reset your machine. Fortunately resource tracking will rescue a situation like this and you can get the system back with everything intact.

TASK DESTRUCTION

Resource tracking also gives us the opportunity to use an even greater feature, which is task destruction. Task destruction is the ability to stop and destroy a task while it is performing its normal processes, and

still leave the system intact. You can stop a program immediately by holding LEFT-AMIGA and DELETE - even in the middle of a video game! This powerful feature uses resource tracking to return your system to the same state that it was in before the program was active. This is quite handy for users and programmers that want to get back to their system as quckly as possible, and obviously is very useful for debugging purposes.

Return To Index

1.12 Copyright Notice

--- COPYRIGHT NOTICE ---

The Games Master System archive may be re-distributed on the condition that its original content is unchanged. Files found in the archive are not available for separate distribution, unless specified otherwise. Some files are the copyright of their respective authors and are subject to their own terms and conditions.

DreamWorld Productions reserves the right to change any of the material within the Games Master System at any time and without notice. Reverse engineering of the Games Master System software or the release of unauthorised programs using or emulating the Games Master System is strictly prohibited.

DreamWorld Productions will not be held responsible for any smoke, volcanic activity, illness, acts of God, earthquakes, or any other harmful incidents caused directly or indirectly through the use of this product.

Return To Index