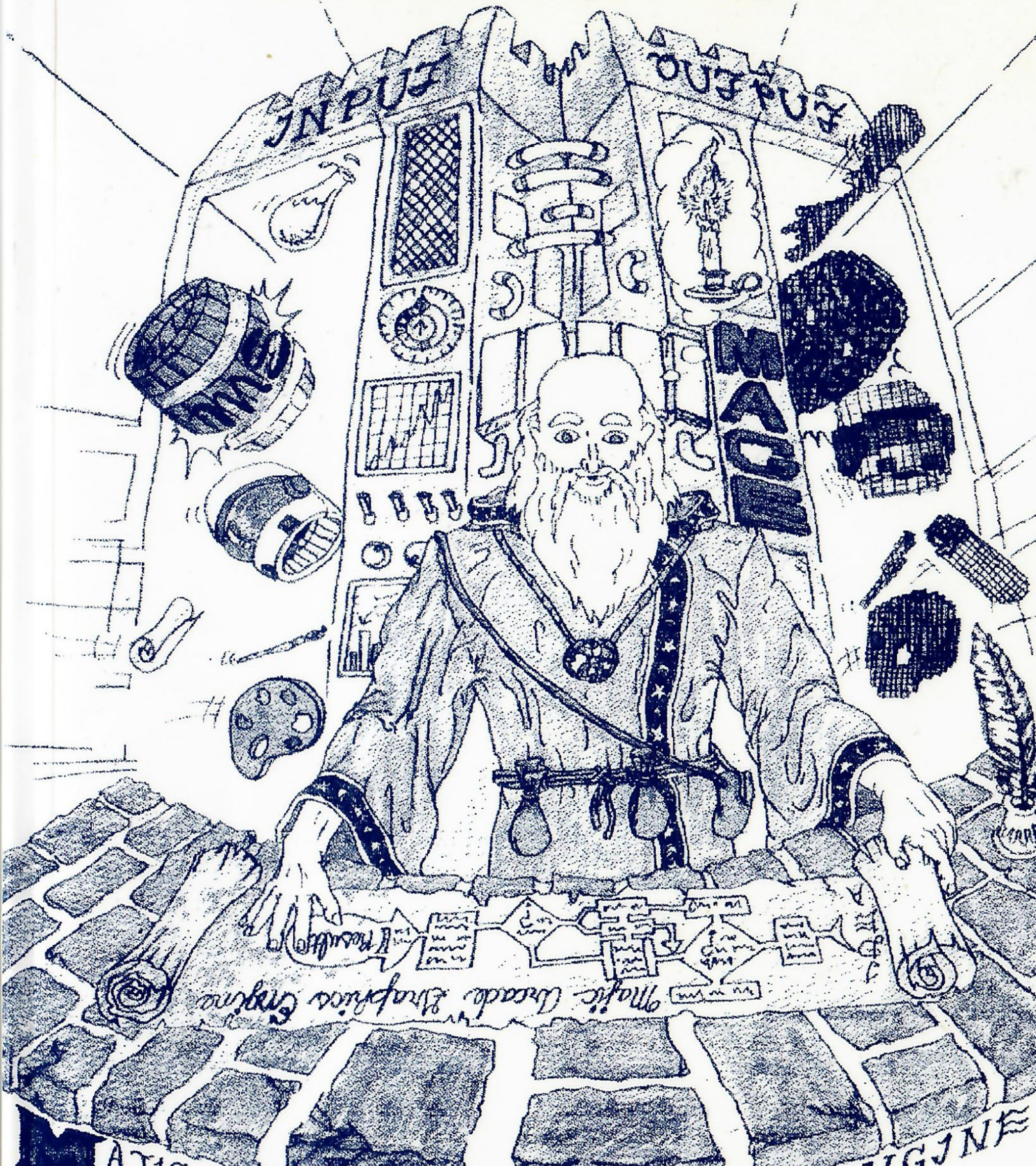
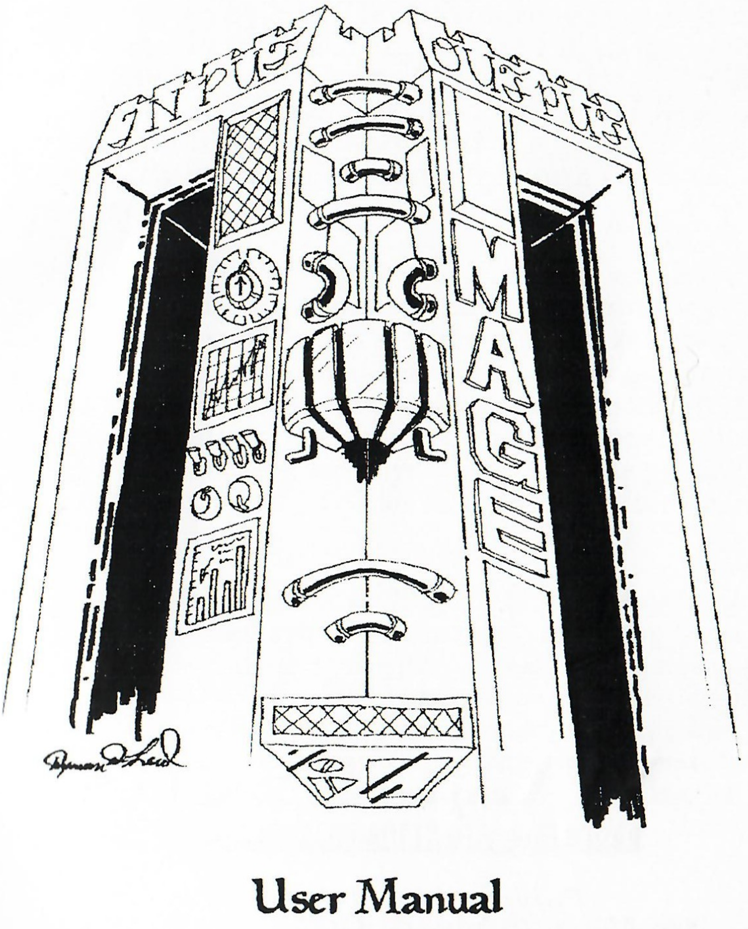
M.A.G.E.™
Majic Arcade Graphics Engine™



User Manual

M.A.G.E.™

Majic Arcade Graphics Engine™



User Manual

Robert Dytmire, Dave Munsie, Larry Scholz and John Stewart

MajicSoft®

"We Put The Majic In The Software!"

Published by MajicSoft, Inc.
348 Meredith Square, Columbia, SC 29223

First Edition, November 1993

The Authors

Robert Dytmire is the Head of the Arcade Division at MajicSoft. He has spent the last two years working with the Majic Arcade Graphics Engine (hereafter referred to as M.A.G.E.) and is qualified in all facets of programming with it. Most of this users manual was written by him. He also contributed approximately 500 of the sprites in the M.A.G.E. package and programmed the enclosed game named "Thurg N Murg". He finished the complete game in less than five days, which attributes to his programming skills and to the power of the M.A.G.E. software. Robert currently resides in Jacksonville, North Carolina. He is twenty-six years old.

Larry Scholz is Head Programmer at MajicSoft and is involved with all aspects of all software developed by our company. Larry has spent the last twenty years programming on many different types of computers from main frames to mini computers and desktop models. He is a fluent programmer in Assembly, C and Basic. He programmed the enclosed game "Sleuth", which he finished in approximately ten days. He is also the proud father of two boys and currently resides in Reseda, California. He is thirty-nine years old.

John Stewart is the founder of MajicSoft. His expertise is in marketing and graphic arts and he enjoys designing games and playing them. He did all of the graphics in the enclosed game "Sleuth". He is the Editor of this manual. When he is not working directly with various programmers at MajicSoft, he is usually found chasing his other hobby - large mouth bass fishing. John and his wife have one beautiful seven year old daughter and currently reside in Columbia, South Carolina. He is thirty-nine years old.

Dave Munsie is the creator of the M.A.G.E. and contributed Chapter 11 - " The Twenty Minute Game". He and his family reside in Haltom, Texas.

Acknowledgements

MajicSoft wishes to thank Wyman Lew for his great art which can be found throughout this manual. He also designed the art work for the box. Wyman is twenty years old and is currently attending college in Castro Valley, California. He is majoring in Electronics Technology and spends most of his free time drawing. The contract to do our box and manual art are his first professional artistic endeavors. Previously his art works have won him Honorable Mention, Imagi - Nation Art Show, Certificate of Merit - Honorable Mention (1984) Strobridge School Art Contest, Sun Gallery Student Artist Ribbon (1987) Sun Gallery Art Show, Certificate of Merit and 1st Place Award (1989) Martin Luther King, Jr. Poster Competition just to mention a few. If you would like to write to him or if you have any questions or comments regarding his work, please send a SASE to:

Wyman Lew
P.O. Box 2811
Castro Valley, CA 94546

MajicSoft also wishes to thank Ari Feldman for his artistic contributions in the main screens of the various programs contained within the M.A.G.E.. He also contributed approximately 600 of the sprites enclosed in M.A.G.E. package. He is presently attending college at the State University College of Oneonta, New York.

A big thank you goes to our Beta Testers: Steve Meredith, Michael Stauffer and David Shumaker.

A special thank you goes out to Steve Meredith for lending us his Atari 800 and "Shamus" game, upon which our "Slueth" game is based. We dedicate "Sleuth" to you Steve!

And for all of the other people that have helped, thank you!!

Table of Contents

Prologue:

When MajicSoft asked me to write this manual I had to ask myself a very big question: "What is the M.A.G.E. system?". Simply put the M.A.G.E. is a tool. It is not a stand alone language like STOS nor is it a compiled extension library like C uses. It is a collection of some of the most amazing Basic routines married around a kernel of Assembly ever introduced into the Atari world.

Because the M.A.G.E. is a tool, and not a stand-alone product, I have not attempted to teach programming in this document. Instead, I have aspired to give a clear overview of each aspect of the M.A.G.E.'s capabilities. Game design theory has been introduced and discussed at length in some sections. The source code provided and the chapters describing individual efforts in game design on the M.A.G.E. go into great detail on how to program a game. However, I am still not convinced that novice users should attempt to use the M.A.G.E..

Users should have the following skills: The ability to program in basic, if possible GFA Basic; the basics of structured programming; and they should know how to utilize the various options that the GFA Basic editor offers such as Inline loading and saving. If you do not know these things, then please use the free copy of GFA Basic provided with the M.A.G.E. and work with the language itself before going on. And if you do not own the GFA manual, please register your copy with GFA Software Technologies and you will receive the manual and warranty card and technical support if needed. You will find a coupon enclosed within the M.A.G.E. package that explains how to register.

Even novice users can quickly become proficient enough to use this tool. The M.A.G.E. is a hammer, not a variable speed diamond drill, and once you've used it once you will never really have to think about it again. In designing this manual I have come to realize an important fact, the user MUST experiment with the system in order to learn it. Some things would take me pages and pages of eye drooping text to explain which will take you minutes, if not seconds to discover yourself when pointed in the right direction.

I feel that this system should raise the quality of software to new heights in the Atari world. If you put forth the effort you have on other, more limited, game creation systems, I have no doubts that the Atari world will be the envy of other software platforms.

So, it is off to finish some selected bits of editing before the manual ships to the printers. I hope it helps you in your endeavor to create great software on the Atari series of computers. And remember, as long as you make a backup of everything, you can not break anything!

Robert W. Dytmire
Arcade Division
MajicSoft, Inc.

INTRODUCTION

Thank you for the purchase of the M.A.G.E. (Majic Arcade Graphics Engine) system. This product is a full development system for creating professional quality games on the Atari ST/STE/TT/Falcon computer systems. Included in this package are all the high quality tools you will need to produce fantastic video games on your machine. The M.A.G.E. game design system does require that you have a working knowledge of basic programming (GFA Basic is preferred) and system operation.

The M.A.G.E. system is a very complex set of software tools and commands that work together to give you the most programming power ever available on the Atari series computers. Because of this, it is somewhat more difficult to master than other, more limited, programming environments. Please take the time right now to turn your system off and READ THE ENTIRE MANUAL. The M.A.G.E. system is a top-down programming environment. Therefore, a clear understanding of the total system usage may not be obtained by reading just a few chapters. Once the entire process is understood, the M.A.G.E. system will be more easily mastered.

The M.A.G.E. system is designed to work with the Atari ST/STE/Mega ST/TT and Falcon compatible low resolution mode (320 x 200 with 16 colors). No other mode will work correctly with the M.A.G.E. system. A minimum system requirement of 512K of memory and a single floppy drive are required to develop with the M.A.G.E., but one megabyte of RAM and a hard drive with at least five megabytes of free space is strongly recommended.

M.A.G.E. puts you in the drivers seat of an advanced game design tool and allows you to effortlessly conform to Atari's published game programming guidelines. This gives your product added compatibility and professionalism.

Before you begin using our software please take the time to read the Software License Agreement and make a backup of all disks included in your system. MajicSoft, Inc. gives you permission to make a single back-up copy for everyday use. It is a good idea to store your originals somewhere safe and cool. Our disks are not copy protected so you should be able to use the standard GEM interface to make your backup.

Software License Agreement

This is a legal agreement between you and MajicSoft Incorporated (hereafter referred to as "MajicSoft"). Please read this Agreement carefully before opening the disk package. Opening the disk package indicates your acceptance of this Agreement. If you do not agree with the terms of this agreement, promptly return the unopended disk package, along with all other associated materials to the supplier of the product for a refund of your purchase price. This original Software License Agreement is your proof of license. Please treat it as valuable property.

1. Grant of License

Subject to the terms set forth in this License, MajicSoft hereby grants you a non-exclusive license to use the M.A.G.E. programs contained in this package (the "Software") only on one computer at a time. You may make one copy of the software soley for backup purposes. At no time may the software be installed in more than one computer at the same time or be used in a netwwork on more than a single station at the same time.

2. Title

Title and ownership rights to the software and its copyrights shall remain with MajicSoft.

3. Use

This software contains valuable trade secrets and copyrighted materials. You agree not to modify, reverse-engineer, decompile, disassemble, copy or adapt the software. The software is licensed for use only in designing, creating, modifying, and running M.A.G.E. computer games.

4. Transfer

This software is licensed solely to you, the Licensee. You may not rent or lease this software. You may transfer your rights under this MajicSoft License Agreement on a permanent basis provided you transfer this License Agreement, the original software, and all accompanying original written materials, retain no copies (in any form), and the recipient agrees to the terms of this agreement.

5. Limitation of Liability

In no event shall MajicSoft or MajicSoft's suppliers be liable to you or any other person for any incidental, collateral, special, or consequential damages of any character, including, without limitaition, damages for loss of profits, loss of data, computer failure or malfunction, claims by any party other than you, or any and all other similar damages of losses, even if MajicSoft, its suppliers, or its agents have been advised of the possibility of such damages. The licensee assumes all risk of using the software and of using any games that are made with the Software, and the risk of anyone using any software that is subsequently distributed by the licensee. Some states do not allow the exclusion of limitation of incidental or consequential damages, so the above limitation may not apply to you.

6. Warranty

The Software is licensed as is, without warranty of any kind whatsoever.

7. General

This License is governed by the laws of the State of South Carolina and represents the entire software license agreement among the parties.

Distributing Your Games

* **Commercial** developers that use ANY portion of the MAGE system MUST obtain a license number from Majicsoft. The fee for this number is $1000 per year payable prior to issuance of the license number and every year thereafter on the issuance date. A lifetime license number may be obtained for a one-time fee of $2500. Companies producing commercial software with the M.A.G.E. system must display this license number in their manual and acknowledge the use of the M.A.G.E. design system.

* **Shareware** - MajicSoft wants to set certain standards for material developed with our system. Shareware authors MUST follow these guidelines to obtain permission and a license number before they may release ANY product.

1. Shareware authors must obtain a license number for EACH PRODUCT they produce from MajicSoft. This is accomplished by sending a copy of the finished product, your name, address

and telephone number to MajicSoft. This license number is provided free of charge by MajicSoft.

2. Shareware software must be able to meet the standards of MajicSoft in playability, function and overall design. Games which do not function correctly, or are buggy, or otherwise reflect badly on the M.A.G.E. system will be rejected and such software may NOT be distributed under any circumstances.

3. Shareware authors must ask for a minimum donation of $5.00 (U.S. dollars or equivalent in other currency) for their software. This must be displayed clearly in the software's documentation. Once you have obtained a license number the shareware author must reflect the license number in his title sequence and his documentation.

* **PD Software** - MajicSoft forbids any user to make available any software written using ANY part of the M.A.G.E. system for free. In other words, NO PD SOFTWARE MAY BE PRODUCED WITH THE M.A.G.E.. The only exception to this are commercial demos of future products which must obtain permission from MajicSoft before release.

Possible Commercial Distribution

MajicSoft may, from time to time, receive a product of such high quality that we may wish to offer you the choice of commercial distribution. Because of this, we reserve the right of first refusal to all non-commercial programmers. This means that you may not send your shareware product to any other company, BBS, Individual, etc... before you obtain a license number from MajicSoft. This applies to sending demos of your product to other companies as well as finished versions. Commercial users may distribute their product as per the commercial license guidelines provided by MajicSoft.

Under no circumstances can the M.A.G.E. GFA Basic source code be given away, sold, copied or published without prior written approval from MajicSoft. This is defined as the GFA source code provided with the M.A.G.E. system. It includes all routines used to communicate with our core kernel, our 68000 assembly code kernel and any other code provided with the M.A.G.E. system. This also includes the tutorial games that are included with the M.A.G.E. system.

Any executable program written using ANY part of the M.A.G.E. must provide the following line of text in the title sequence of the program and in the beginning of any accompanying documentation:

"Programmed Using The M.A.G.E. - ©1993 MajicSoft, Inc.."

INSTALLING THE M.A.G.E. SYSTEM

Floppy Drive Users - Simply make a backup of each disk (they are not copy protected) and work with your backups. You may ignore or delete the INSTALL.PRG on your backups.

Hard Drive Users - Make sure you have a partition with a minimum of 5 megabytes of free memory. Place the M.A.G.E. Master Disk into drive "A" and call up its directory. Double click on the INSTALL.PRG. The program will prompt you for any info you need to input and/or disks to swap. It's that easy!

THE PURPOSE OF THIS MANUAL

The purpose of this manual is to overview game design using the M.A.G.E. system. It is not intended to teach you how to use GFA Basic nor is it written for the complete novice. The M.A.G.E. is an advanced, yet easy to use, game creation tool that assumes programming knowledge. If you are new to programming in general it is suggested that you read through your programming language's manual and that you become familiar with GFA Basic in particular before continuing. Intermediate and advanced users should have no problems understanding the entire M.A.G.E. system.

Preface - The Basics

This section will overview the different basic concepts that the M.A.G.E. system embraces. It will also give you an idea of what level of knowledge is necessary to use this product most effectively.

Variable Types

The M.A.G.E. game design system uses three types of variables when communicating with GFA Basic. These are the byte(|), Word(&) and long word(%) variables. When a command is explained in the manual the variable types it needs will be outlined. It is vital that the correct types of variables be used or a crash may result. Be aware of this because most of the problems that beta testers have encountered are in the form of variables passing errors.

Overview of M.A.G.E. Components

The M.A.G.E. system uses four editors to prepare data for the M.A.G.E. run time code. These programs encode your graphics, maps and scripts into the highly specific data the M.A.G.E. requires. These editors are as follows:

Mage Character Editor - Prepares raw graphics you draw or select for use as sprites or characters.
Mage Animation Editor - Prepares scripts that the M.A.G.E. uses for movement and animation during game execution.
Mage Map Editor - Allows for custom-background creation from a character set.
Mage Inline Maker - Converts the above files for use in INLINE statements (an advanced feature of the M.A.G.E. development system).

The M.A.G.E. Shell

The M.A.G.E. shell is a whole set of GFA Basic codes specifically designed to work with the M.A.G.E. 68000 assembly core. It includes all necessary loops, structures and commands to complete an entire game. The MAGE16.INL, MAGE32.INL, MAGEBLT.INL and MAGEICE.INL cores must be inserted into the GFA code as **inline** statements.

The M.A.G.E. Sprites

Over 1000 sprites are included for your use with the M.A.G.E. game design system They are drawn in M.A.G.E. sprite and Character sizes and are completely palette compatible. Never before has such a professional and extensive library of game sprites been made available to programmers.

The Chip Music Replay Interface

Based on a popular PD mod player, our GFA Basic interface allows for exacting control over the ST's Yamaha sound chip like only the professionals had before! Rich three channel chip songs and sound effects are a single command away!

Overview of the M.A.G.E. Game Creation Steps

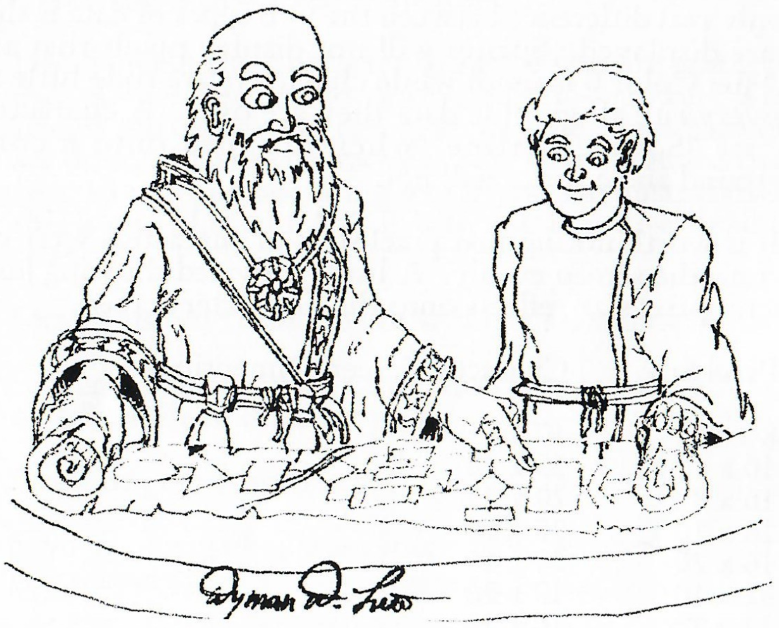
When developing software with the M.A.G.E. you will need to follow this general course of events:

1. Outline your game.
2. Create your graphics.
3. Use Character Editor to convert graphics into Sprites and Fonts.
4. Prepare animation scripts in the Animation Editor.
5. Prepare any backgrounds in the Map Editor.
6. Program your game.

As you can see, there is a lot of preparation involved with the M.A.G.E.. This is because the M.A.G.E. is not a language but a collection of very specific and very powerful commands that expect the data encountered to already be in certain forms. Half of writing games with the M.A.G.E. will be working inside the various editors before hand. The advantage to this is speed, ease of programming and better use of memory. We will talk more about the specifics of designing a game in future chapters.

MajicSoft®

"We Put The Majic In The Software!"



Chapter 1. The Editors

The purpose of this chapter is to fully explain each of the M.A.G.E. system editors. You will be introduced to the concept of a Character Screen, a Character Map, a Sprite Image Bank and a Character Image Bank.

The Character Editor

A. Introduction to the Character Screen

The character editor is probably the most used tool in the M.A.G.E. development system. This editor is designed to produce two specific types of data. The **sprite images** or *.SP1 files are the images you will use to represent moving things. The

characters or *.CP1 files are the background graphics and fonts. The only real difference between the two types of data is the way they are displayed. Sprites will not display pixels that are not "ON" (ie: Color 0 is used) while characters are rude little things that overwrite all graphic data they are over. A character will show its "Square outline" when dropped onto a complex background and a sprite will not.

It is worth noting that pixel sizes of characters were chosen to fit into the screen evenly. A list is provided showing just how character pixel size reflects onto the Character Screen:

Pixel Size	Character Screen Dimensions
8 x 8	40 x 25
10 x 10	32 x 20
16 x 8	20 x 25
16 x 10	20 x 20
16 x 20	20 x 10
32 x 10	10 x 20
32 x 20	10 x 10

We will talk more about the Character Screen in chapter IV. As you can see, the smaller the character size the more data you can get onto the screen...although you may not always want to use small character sizes.

Both a sprite and character set are limited to 200 cells each. This number was chosen as a good balance between size and speed. If more sprites or characters are desired, the M.A.G.E. system is designed to allow for multiple "Banks" of sprites and characters. Indeed, beta testers did not find 200 sprites much of a limitation, as most games they designed failed to fill even a single sprite bank.

Steps to creating a character or sprite set:

1) Decide what size character or sprite you are going to use. Remember that for sprites bigger = slower. The size of the background graphics depends on what you are doing with the characters. It is recommended that 16 x 10 be your operating size if you are unsure since this is one of the M.A.G.E.'s fastest and most flexible sizes.

2) Select this size from the NEW menu selection.

3) Either define you palette manually or load in a *.PI1 or *.PC1 file to do it for you.

4) Draw or grab your images.

5) Save the image bank as either .SP1 (Sprites) or .CP1 (Characters).

B. Character Editor Command Summary

Activate the MS_CHAR.PRG program to start your editor. The program will appear and you will get a menu bar across the screen. This works very much like a GEM menu bar except you must keep the mouse button pressed to access the functions in any given header. Two boxes also appear on the screen. The left hand box is the image selection box and allows you to select an image for editing or for use in some of the commands listed below. Because of size limitations, not all images may be displayed at once; you may page through them with the V1-X title. The left hand box is the editing box. You may draw (Left button) and erase (Right button) inside this box. The palette is displayed along the bottom of the screen. You may select the current drawing color by clicking the left mouse button while putting the mouse pointer over the desired color block. The current drawing color is reflected in the mouse pointer (it will change to that color).

Here is an overview of the Character Editor's commands and functions:

Title: LOAD

?P1 -	This allows you to load in a .SP1 or .CP1 set for editing.
.PI1 -	Load in a non-compressed DEGAS format picture.
.PC1 -	Load in a compressed DEGAS format picture.
.NEO -	Load in a NEO format picture
.32K -	Load in a 32K chunk of data...also known as raw data.
.PAL -	Install a saved 16 word color palette (see GFA XBIOS calls for format).

Note: You will replace your palette when loading in a PI1, PC1, NEO, CP1 or SP1 file.

Title: SAVE

?P1 -	Saves either a SP1 (sprite) or a CP1 (Character) bank. If an illegal extension is given then a CP1 set will be saved.
.PI1 -	Save current background as a DEGAS non-compressed picture.
32K -	Save the background as a 32K memory segment. No palette is saved!
.PAL -	Save the current palette.

Title: COPY

This function copies the current cell selected into the next cell selected.

Title: SWAP

This function exchanges the currently selected cell with the next selected cell.

Title: GRAB

These selections all refer to the size of the image you wish to grab (in cells) from the background screen. The 1 x 1 means a single cell while the 3 x 3 means a 3 cell by 3 cell block will be grabbed (good for grabbing large "Boss" type creatures). Once you select a size the menu will disappear leaving you with the background screen. A square and the mouse will now be displayed. The square represents the area the cells occupy (useful for lining up your image). Place the square around the image you wish to grab and click the left mouse button. The screen will shift back to the menu and you will notice that your captured image is now inside the left hand display. Place the image(s) in the cell(s) you want to occupy. Click the left mouse button to confirm or the right button to abort the operation.

Title: PASTE

Allows you to paste an image (on character cell boundaries only!) onto the background. This is useful for creating "Mock Ups" of backgrounds and/or game screens to check your work for flaws. It can also give you an idea of what the images will look like once they are assembled.

Title: FX

Flip H	- Flip Image Horizontally.
Flip V	- Flip image vertically.
Bold	- Attempts to make image thicker.
Scroll	- These commands move the image (with overlap) around its cell.
Undo	- Unlike most undo functions this one simply redraws the original image in the cell.

Offset Grab - Possibly the most confusing and powerful commands in the Character Editor.

Here is how it works:

It assumes that you have created a grid of images and have loaded them into the background. What this function will now do is load them all into the bank at once (eliminating single grabs!!). All images grabbed must be in a grid like formation (spacing must be the same or it will not work). When selected, the function will ask you for the starting upper left pixel of the first image to grab. It will then ask you for the pixels between images, **not** including the pixels of the images themselves. If you have drawn a grid for your images (like the Thurg .PC1 screens), then you will have a 1 pixel spacing between images in both directions. It will then ask you how many images across and down you wish to grab (answer these as you have drawn your next grid). It will then ask you which cell to start placing the images into. After this data has been entered the editor will give you a preview of what is going to be grabbed. Press Y to confirm this grab or return to abort the whole thing. This powerful function is a little strange at first, but with a little practice you will wonder why every sprite editor does not have something similar.

Title: V-1

Clicking on this will page through your image bank.

Title: NEW

Allows you to set up a new image bank. Unlike some other systems all images in a given bank must be the same size. Unlike other systems, the M.A.G.E. allows you to combine multiple images easily.

Title: QUIT

Returns you to the desktop.

Additional commands/notes:

The **[Help]** key will present you with a list of keyboard shortcuts to the above commands. Not all commands have shortcuts.

Pressing P will bring up the Palette Editor. This editor is very simple to use. Simply click on the arrows to change the selected colors. Default will reset the colors to the M.A.G.E. system default.

[Alternate]+[R] will allow you to run another program without leaving the editor. This is great for floppy based developers.

C. Tips for using the Character Editor:

- Plan your characters around the ASCII character set.
- Remember to leave character 32 blank or filled with your background fill because the map editor uses this character for blank fills.
- Remember that small sprites generally move faster than big ones.
- Use your own graphic program to design the images because it is probably more powerful for actual drawing.
- Look at our examples on the Source disk. Play with the full screen grab functions.

The Animation Editor

The M.A.G.E. animation editor is a powerful tool used to create animations, patterns and offset movement sequences for your game. Once created, the resulting file allows for effortless animation and pattern movement, such as never before seen in GFA Basic! The animation editor creates 4 forms of data for the M.A.G.E. system to process. These are as follows:

A. Data Types the M.A.G.E. Uses

Animations - A sequence (usually repeating) that will be played back when called. This is a collection of sprite frames chained together to create the illusion of movement. Walking is usually accomplished by stringing together two separate sprite images.

Patterns - A fixed movement path that you draw on the screen. A sprite can then be told to follow the pattern for quick and easy movement.

Offsets - Similar to patterns except they do not represent actual coordinates but directions to move each step. The advantage to this is that an offset can start from any position on the screen, while a pattern is fixed. A "Galaxian" type game would use this form of movement.

Offsetf - Much like the offset but with the additional advantage of being able to follow another sprite while performing an offset. By drawing a circle you could assign a sprite to orbit another sprite or you could launch a sprite on an offset based on an invisible sprite ("Insectroid" uses this trick).

B. Animation Editor Command Summary

Activate the MS_ANIM.PRG to activate the M.A.G.E. Animator. BE WARNED: The Animator requires an .SP1 file to be already created in order to make animations. The Animator will **not** work with .CP1 files (they are for the Map Editor).

You will be presented with a simple menu with 5 main areas and several sub-sections. To select an option, simply point and click. Here is a overview of the Animator's menu:

Area: SP1

LOAD	- Load a previously saved .SP1 file. This erases the current records!
SAVE	- Saves out a .SP1 file for engine use. Also activates the printed report option.
INFO	- Program information.

Area: Load

SP1	- Load in a sprite file created in the Character editor. Should be your first step.
PC1	- Load in a compressed DEGAS background screen. Good for viewing your animations over a background.
PI1	- Loads a non-compressed DEGAS screen into the background.

Area: Anims

The animation slot currently under edit. Clicking the left mouse button on an arrow changes the slot very quickly. Clicking the right button on an arrow will single step the slot number. This also points to the last animation to be saved. Be careful of this because you can save a partial list if you forget to move the pointer back.

MAKE	- Create a new animation in this slot (old animation is erased).
PLAY	- Play back the selected animation slot.

Area: Patts

The current pattern being edited. Uses the same functions as the Anims area.

MAKE	-	Create a new pattern in this slot.
PLAY	-	Play back pattern using currently selected animation.

Area: Offset

The current Offset (or Offsetf) being edited. Uses the same functions as the "Anims" area.

MAKE -	Create a new offset in this slot. This function has another selection asking for offset and offsetf. Review the descriptions above for the type of offset to create.
PLAY -	Plays back the offset. Offsetf can look a bit strange at first.

Keyboard Commands:

ESC	- Exit to the desktop.
HELP	- Displays a small help screen.
[ALT]+ [R]	- Run another program without exiting.

Notes:

- The Animations, Patterns and Offsets all use a 1 based animation notation. Slot 0 does not exist.
- Playback of Patterns and Offsets will use the currently selected animation.

C. Creating Animations:

Once you have loaded in your .SP1 file (the Animator must have something to animate!) you can begin to create your animations. To start, select the up arrow in the ANIM section. This will select slot 1. Note that you can not go any higher...this is because you have not filled slot 1 with something. Once you do, you will be able to move to slot 2. Now Select MAKE. The program will now display the first group of sprites in the bank. To see more groups in the bank press return. On the bottom of the display there will be a report telling you which frame you are on, which sprite you are selecting and what group you are looking at. To add a frame to the animation simply highlight the desired sprite and click the left mouse button. When you are finished, click the right mouse button or ESC. This will bring up a description request box. This box is a rather nice feature of the Animator in that you can enter a description of the animation for later printout. No more trying to remember which animation is which or hassling with lots of scrap paper. You can now see your animation by selecting PLAY from the ANIM area. The Animator will show you a large selection of speeds and directions so that you can see the animation in action.

Tips for creating animations:

- To make animations slower, click on the same frame multiple times.
- There can be a maximum of 101 frames per animation and 100 animations total.
- Pong effects can be achieved by selecting the animation and then selecting it in reverse order. In other words enter the whole pong loop manually.
- **Always** play back your animation, it is possible you could make a mistake and you want to catch it here...not later while you are programming.

D. Creating Patterns:

Mouse:

Creating a pattern uses basically the same process as the animation area. Select your slot and select MAKE. A sprite will replace your mouse pointer on the background screen. The first time you click a mouse button tells the computer to start recording your pattern. Holding down the left button will record movement continuously, regardless of mouse position (ie: if the mouse stops moving your pattern will record a pause at that spot). Holding down the right button will cause the program to record the pattern only when the mouse position changes. No recording will happen if no mouse button is pressed (allowing you to "jump" around the screen). When you are finished with your pattern press the ESC key. This will activate a description box, allowing you to type in a brief description of what each pattern is. Experiment with both types of recordings by using PLAY to see your results. In any case, each pattern may hold no more than 500 recording segments (a bell will sound when you reach 450). The animator can record no more than 100 patterns.

Keyboard:

Very much like the mouse input method except the arrow keys control input and the **space bar** acts as an enter key. The space bar tells the program to record a single segment. Shift and Arrow keys shift the sprite two pixels instead of one. This method of entry is much slower than the mouse method and allows a higher degree of precision. This is necessary for some patterns.

Tips for creating patterns:

- You can alter the sprite's position relative to the mouse pointer by using the arrow keys (which allows you to draw off the screen). Pressing UNDO will set it back to its original position.
- The Right button method usually results in a smoother pattern playback.
- Feel free to mix methods!
- **Always** test your patterns.

E. Creating Offsets:

Offsets are entered in exactly the same way as patterns. Read the Creating Patterns section to learn all the nasty details. What differs with an offset is how you use it. The starting position of the offset is only relative. That is, unlike a pattern which will move **exactly** as you draw it, the Offset will move just as you draw it but not necessarily in the same spots. It is like drawing a jump pattern for your main character. Every time the player jumps you canny an offset. The jump offset will play back exactly as drawn originally, but it will start wherever you tell it to. This allows for more flexibility in planned movement. The maximum is 500 segments per Offset/OffsetF. The Animator will hold up to 100 frames of these segments.

Tips for creating Offsets:

- Offsets need to be more accurate than patterns, especially if they repeat.
- The sprite that is laid down when your first position is selected can be very helpful. If you select the very last slot (an empty one) there will be no animation. Now you can **exactly** line up your first and last position segment in the Offset.
- **Always** play back your offsets...if they go crazy then you do not have the first and last position lined up.

F. Creating OffsetFs:

Creating an OffsetF is very close to creating an offset. The major difference is when you first click the mouse button or space bar, you will create a reference sprite. The next click of the mouse or space bar is the signal to begin recording. Imagine the reference sprite as the sprite your OffsetF is going to follow. Now draw (using the methods described in the Pattern section) the path that you want the OffsetF to use. With this you could draw a circle around the reference sprite (items would orbit the reference sprite) or up and down next to the sprite (for shields), etc... I am not going to kid you here...this one may take some experimenting to fully understand, but it is well worth it!

Hints for creating OffsetF's:

- Be careful. This one sounds like fun but needs careful entry to work most effectively.
- **Always** check your OffsetF's carefully!

THE MAP EDITOR

A. Introduction to the Character Map

Have you ever wondered how certain games get so much background data into such a small program? Well, the way it is done is to create an interlocking "Lego Set" of character cells and instead of storing actual image data, store a map of the cells. One byte of data now represents up to 320 bytes of data! In order to make it easy to construct such maps we have provided you with the M.A.G.E. Map Editor. To use this system you must first create a set of character cells with the Character Editor (a .CP1 file).

B. Map Editor Commands

Activate the MS_MAP.PRG to start your session. When activated the Map Editor will display a true GEM menu bar. The first thing you need to do is to load in a character set. No other function in the program will work until you do. Once this is done you can select from one of the following:

Title: DESK

ABOUT - Information about the program.

Title: MAP

NEW	- Clear the current map and pointers.
LOAD	- Load a .CP1 file or a .MAP file.
SAVE	- Save your MAP (area defined by the F8 and F9 settings).
EDIT	- Invoke the Map Editor.
QUIT	- Return to desktop.

Title: SCREEN

Save as .PI1 - Save the current displayed portion of the map as a non-compressed DEGAS compatible file.

The **[HELP]** key will show you the keyboard shortcuts for the above menu functions.

The Map Editor:

When first invoked the map editor will display the "Character Selection Menu". This will not appear as much of a menu (and may look like a blank screen if your first few character slots are empty). Whenever the Character Selection Menu is active, several things occur: First, the top line of the screen becomes a scrolling representation of the current character map. By moving the mouse to the extreme left or right of the screen, you cause the characters to scroll in their respective directions. By pointing to the character you want and clicking the left mouse button, you make that character your current drawing pen. You may also point directly into the current section of map to select your new pen. Clicking the right mouse button while this menu is active will return you to the GEM menu and exit you from the actual map editor. Once you select a pen character you may draw with the left button and erase with the right button **(character 32 is used for erase)**. The map editor can handle screens up to 16 x 16 per map. Several keys are used to invoke different pointers and functions during drawing. Pressing the HELP key will invoke a small menu to remind you of these commands. They are as follows:

F1 - Set top left of block to current pointer position.
F2 - Set bottom right of block to current pointer position. MUST be greater than F1 above.
F3 - Copy block to current position. You may NOT overwrite your defined block area.
F4 - Fill current screen with character 32 (ie: Clear the screen).
F5 - Auto scroll on or off. Auto scroll will change the arrow key functions to a single cell width for scrolling, instead of the default screen boundary scrolling.
F6 - Move to the top left of the map.
F7 - Move to the bottom right of the map.
F8 - Set the top left of the save area
F9 - Set the bottom right of the save area.
F10 - Search and replace (screen area only). To use this you will replace the character now used as the pen with the next selection (this command invokes the Character Selection Menu).

Arrow keys - Page up, page down, left or right or scroll up or down or left or right if Auto scroll is invoked.

Maps can hold many types of game screens. "Pac-Man" type games use maps for mazes, while platform games like "Mario Brothers" and "Thurg N Murg" use them for actual play fields. Games like "Mario World" and "Gods" use scrolling maps to create worlds bigger than a single screen. No matter which way you choose to use your maps, the editor can provide you with a quick and easy way to create your game world.

C. Tips on Using the Map Editor:

- Try to keep character cell 32 clear since the Map Editor uses it for erase and clear.
- Mark the top left corner of your map with F8 as soon as you start editing.
- Search and replace is a powerful function for custom CLS commands.
- Note how different character sizes make for fast or slow scrolling. Try out several sizes to get a good feel for how your character scrolling is going to work in your game.
- The Map save area must be at least a 1 x 1 screen in order to save. No smaller sections are allowed.

THE INLINE MAKER

A. Introduction to the Inline Maker

Note: The Inline Maker is an advanced feature of the M.A.G.E. system and you may not want to read this section right off. If you are new to the M.A.G.E. system then our recommendation is to skip this section until you have mastered the other editors. The implementation of this editor can be confusing to novice users. It is not necessary to use this editor to complete a fully functional game with the M.A.G.E. system.

Ok, you have been warned. Now on to the editor that will give your program that professional touch. In addition, it will help you conform more closely to Atari's own published specifications on game design (ie: Make it a single file if possible). You will need to use the shareware program JAMPACK4.PRG to compress your files. It can be found on the M.A.G.E. Extras Disk. MajicSoft supports shareware authors and we sincerely hope that you will too. Please register your copy.

The Inline maker prepares your data for insertion into your program as an **inline** statement. This means you should be pretty sure you are not going to want to work on that data again because you will have to uncompress the data if you need to edit it. This should really be the last step in completing your game.

B. Inline Maker Commands

Activate the MS_INLIN.PRG to invoke the Inline Maker. The Inline Maker works in a three step process. First the data in your .MAP, .CP1, .SP1 or .GPA will be stripped of their headers. Next, the raw data is compressed by Jampacker (which you must exit the Inline Maker to do). The last step is to combine the header and the compressed raw data into a final **inline** form. You can then **inline** this data and use the special sprite and character installer routines provided in the M.A.G.E. GFA Basic shell to activate the data banks. WARNING: Use ICE Compression at 1024 ratio with Data selected and Flash and Heading deactivated.

Here is an overview of the commands available. The actual design process is explained in the Command Descriptions.

Title: DESK

ABOUT - Information about the program.

Title: SP1/CP1

Here is where you create the Sprite and Character Inlines. The first thing to do is select make ?NL/?CE.

MAKE ?NL/?CE - This takes a sprite file (.SP1) or character file (.CP1) and divides it into two separate parts. The ?NL file created is the header info and you should never tamper with it. The ?CE is the raw data that you must compress with the "Jampacker" program. Exit the program and compress the file using the specifications mentioned above. When saving the compressed file, simply overwrite the original ?CE file with the new data. Now go back to the Inline Maker.

MAKE ?GP - This command takes the header and compressed raw data and saves out a CGP or SGP file that is ready to be inlined. When selected, point to the ?NL file you compressed in the above process and the M.A.G.E. Inline Maker will do the rest. The old files will be erased and a single ?GP file will be all that remains to be inlined.

Title: MAP

MAKE MNL/MCE - This program will take a selected .MAP file and split it into two parts. The .MNL file is the header and you need to do nothing with it. The .MCE file is the raw map data and this is what you need to compress following the rules listed above. Once you have done this you can use the next command.

MAKE MGP - This takes the .MNL header file and connects it to the compressed .MCE file. The result is a single .MGP file for you to inline into your program.

Title: GPA

MAKE GCE - Makes a single file with the extension GCE. Compress this using the rules stated earlier and then come back to the Inline Maker. You can now use the next command.

MAKE GGP - Converts the GCE file into a CGP file which is ready to be inlined into your program.

Title: HELP

GENERAL - Gives you the general packing rules to follow when making inline data.

FILES - Describes the file types used by the M.A.G.E. Inline Maker.

C. Tips for using the Inline Maker:

- If a GPA file is greater than 32,000 bytes long **before** it is compressed, it can **not** be made into an **inline.**
- Do not Panic! The file types are not that hard to remember.
- Do one file type at a time until you become familiar with this system or you may lose track of what you are doing.
- This seems like a lot of effort but it will make your programs shine when all you have is a single .PRG file and someone else's software has 20 or 30 extension files!

Chapter 2. The MS Design Shell

A. Introduction to the MS Design Shell

This chapter will deal with an overview of the M.A.G.E.'s GFA Basic interface. It is not an extensive overview of the M.A.G.E. commands, but rather a look into how we have designed the programming environment.

B. What is the M.A.G.E. Shell?

The MS Design shell is the GFA Basic interface you will work with. It is broken down into two basic areas: The command area is where all commands have been built for you.

You should never have to enter this area of the code. The second area is the user area. This part of the code is pre-constructed to allow you to easily design a game. You will work extensively with the user area.

C. Overview of the Steps - Getting Started

To load the shell, activate GFA Basic 3.5 or higher and load in the file MS_MAGE.GFA. The user area is broken into two parts: The upper part of the code (which handles the initial machine interface) and the lower part of the code (where the actual game programming occurs). The user shell will now be described in the order in which it is laid out. Follow along on your computer if you wish...it will help.

STEP 1 - Step one is an automated process. It automatically puts the machine into the correct graphics mode (low resolution) on any machine.

STEP 2 - The correct **inline** data must be loaded into the slots. If you are working with 16 bit wide sprites or characters load in the MAGE16.INL. If you are using 32 bit wide sprites or characters then load in the MAGE32.INL. You can **not** mix the two types. If you are going to use the advanced blit features of the M.A.G.E. system then load in the MAGEBLT.INL. If you are going to use de-compression routines you must load in the MAGEICE.INL. Press HELP and then "L" with the cursor over the correct inline area to load in the correct core kernel. You will also need to tell the M.A.G.E. which type of blits you are using. Follow the example shown in the code.

Step 3 - Place any **inline** data your program needs in this area. This includes data created with the Inline Maker. You will need to know the length of the file in order to reserve a properly sized inline buffer. Music, sound effects and compressed picture data will also be placed here. We will discuss exactly how you embed all these types of data in the command listing. For now, it is enough to be aware of the fact that such data belongs here.

Step 4 - Call the M.A.G.E. Program_Init routine. Send it TRUE if you are using GEM (Not recommended in a video game environment) or FALSE if you are not planning to use it. Novice M.A.G.E. users should leave MS_Spritedump& set to true. This step also does things such as activate the joystick in slot 0 and shut down the mouse. It also forces a mini-reset of the screen display banks.

Step 5 - This is the BIG call. The MS_Graphics_engine_init(?) routine sets up the M.A.G.E. core for processing your data. You still have to load in the Sprites, Maps, Characters and Animations, but this call gets everything ready to rock and roll!

Step 6 - Call the Play_Game routine. The Play_Game routine is the last procedure in the M.A.G.E. Shell. Press **Control-Z** to get there quick. From here on out it is show time!

Step 7 - End the program. Re-activates the mouse, restores the original screen position, colors and resolution. It also frees up the memory you were using and flushes your program from memory. This is the nice way for a game to behave. All automatic! The F7 bit is used to run an external GFA file.

That is all there is to starting your own game! Most of it is automatically handled by the M.A.G.E. Shell. Now skip the GFA command core and find the Play_Game routine (press **Control-Z** to do so).

PLAY GAME:

Play_Game is probably the most important section of code in the shell. From here you initialize your game, handle the title screen, run through the main loop and handle the "Game Over" routines. This is all laid out very carefully and in a specific order. We will go over the routines now:

Ms_screeninit(0) allows you to reserve full 32,032 byte screen buffers for extra items. The number passed is one less than the number of screen reserved (-1 = no screens). These screens are in addition to the MS_Logical%, MS_Physical% and MS_Background% screens.

Next, we load in any data from disk storage. You should **not** modify the order of loading. We really want the MS_LoadGPA to go first and the rest of the loads do not matter. Give the path to the data with each call.

Next, we load in any **inline** data. This is the preferred method of loading data because it cuts down on external files. Adr% is the **inline** you used to load your data into - in step 3.

The next step is to place any game data into your program. Tables are the most often used form of data. We suggest that you place this data into a separate procedure with its own name. By doing so, you will be able to close the procedure and hide it. This makes moving around your code much easier.

Now we declare our arrays and global variables. Again, we suggest that you put these into a separate procedure. See the "Thurg N Murg" source code for an example.

Set up any variables that you declare only **once** per program execution. These are things like screen boundaries and internal pointers.

You will now see the MS_VSYNC(0) command. This tells the program to prepare for page flipping. You **must** do this if you are writing a game that uses page flipping (ie: ALL Games!).

The first loop we encounter is the main program loop. It will keep cycling through the following sequences until the user exits the program:

- Title Screen - This routine handles your title screen sequence.
- Init Game - Starts the game if the user requests it.
- Store the working pallet for quick restoration.
- Enter the game execution loop. This loop is the main loop during the operation of your game. It is the "Master Brain" behind the smooth execution of a program. This loop does the following:

  1) Resets the Vertical Blanks counter.
  2) Asks you to update all moving "Players".
  3) Page Flips (the last screen draw is displayed and the background is written to MS_Logical%).
  4) Requests all collision checks be handled. This includes all background collision checking.
  5) Calls the M.A.G.E.. This plots all sprites, stars, does color cycling and flash effects.
  6) Calls the keyboard input handler if required.
  7) Waits until the number of Vsyncs you have requested have expired.
  8) Shows Frames per Second the game is currently running.
  9) May go into single step mode if requested.

This loop continues until the player is out of lives or an exit is requested.

D. VSYNCS and FPS Explained

We had better get the whole definition of Vsyncs and FPS out of the way right now because these are probably the most important terms used in this system. We will also tell you about the different screens the M.A.G.E. uses to page flip. Do not worry if this sounds a little confusing. You will fully understand it by the time you have finished your first game.

VSYNCS (pronounced Vee-Sinks) - Your television or monitor runs at either 60Hz or 50Hz. The Hz means cycles per second. In this case it also means picture frames drawn per second. Your monitor does not care how you are updating the screen because it needs data **now** and your computer will feed it regardless of your wishes. This data comes from the RAM starting at MS_Physical%. Since we page flip (change the value of MS_Physical%) it is important that we do this only when the monitor's raster beam is at the top left of the screen (getting ready to draw a new frame). That is where the Vsync signal comes in. A Vsync signal is generated by the computer which tells us that the raster pen is at the bottom right hand of the screen (the last pixel) and is now moving up to redraw the screen. Thus the command MS_Vsync(20,0) waits for a Vsync signal from the computer and then **very** quickly flips the MS_Logical% and MS_Physical% screen addresses. It does this so fast that it happens before the raster beam gets up to the next pixel! What you get from all this is a flicker free graphics image.

If you change your page flipping routine to MS_Vsync (20,32000), you will get screen changes whenever you hit this command. The new data will be displayed starting wherever the monitor's screen pen happens to be at the time. In addition, VSyncs measure how fast a game is running. A typical M.A.G.E. game runs at 3 VSyncs. That is, the monitor draws three screens before we are ready to page flip. This is alright since 20 frames per second (FPS) is quite fast (60 VSyncs divided by 3 screen redraws per page flip).

You can get M.A.G.E. games to easily run as fast as one VSync (60 FPS) if you use only a few sprites and do not re-draw the background. The command MS_WaitVsync(X) is put in to make your game run at a set speed. By setting X to the fastest speed you want your game to move (it should be the average of what it **can** move) you can make your game appear to run very smoothly. The whole upshot of all this is that we will be using VSyncs and FPS in future commands and game descriptions and we want you to understand this before hand.

E. Description of Individual Routines

Now we will describe some of the secondary routines that the PLAY_GAME main loop calls:

Title Screen:

This procedure is where you control the entire title screen sequence, which is normally more than one screen or effect. The bare bones of this routine is enough to get you going and the code is pretty self-explanatory. Look at "Thurg N Murg's" title screen sequence to see how you can run a more complex set of screens. This routine is a "Fiddle" routine in that you can make it as simple or as complex as you wish. We have made it so that Fire or ESC will exit this procedure. Use this until you achieve a higher degree of mastery with the M.A.G.E. system.

Init Game:

This procedure starts your game. It is called **every** time you re-start the game. It sets things like the starting player's lives, level and score. It resets the sprites and gets your screen boundaries setup correctly. It is also where you first call Init Level, Init Sprites and Init Game Screen.

Init Level:

In this routine you will set up all the variables that are necessary to begin a level. Some things that get set are the number of bad guys, the possible locations of game items (which change) and maybe even some complex tables (for advanced programmers).

Init Sprites:

This routine is where you should position all of your sprites that appear at the beginning of a level. This is all you do here.

Init Game Screen:

This is where you draw your backgrounds, character maps or character displays for the beginning of the level. It is wise to copy it to the Logical Screen since it will be displayed once before a re-draw is preformed.

Update Players:

This procedure is where you will control all of the player and monster movements in the game. This procedure usually only calls subroutines specifically designed to control each sprite's movement. This routine will take advantage of the **s1** through **s10** automatic movement controls.

Collision Check:

Now that we have updated the players' positions, it is time to see what collisions have taken place. Two major types of collisions should be checked for. They are character collisions (see the Character Map chapter) and sprite collisions (see the Sprite Chapter).

Keyboard Control:

This directs functions related to key press actions. It is already set up to handle function keys and to restart the game on the first ESC key press and quits to desktop on the second ESC key press. Pause is also handled and so is the VSync rate keys. In other words, you only need to insert program-specific modules here to make things work. We would like for you to keep the keyboard and function key assignments standard for all M.A.G.E. games whenever possible.

Below are some routines that you will **not** want in your finished game:

FC7 - This procedure runs an external PRG (not something to do in a game!)

FC8 - You definitely do not want to dump your program's data onto someone's printer once it is finished.

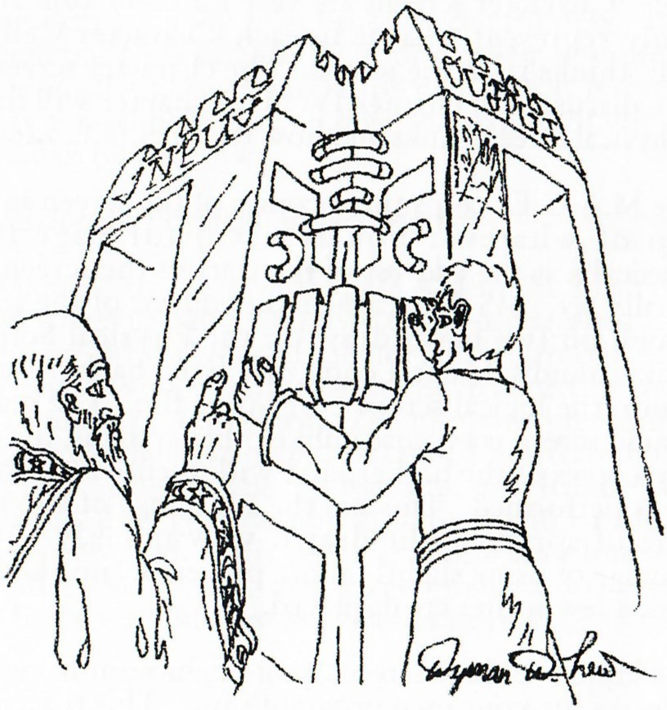
MS_ShowFPS -	The number in the corner ruins most games.
MS_Stepmode -	Most players should not single step through your game.

F. Tips for Using the M.A.G.E. Shell:

- Do not worry that your first few games do not use all of the procedures. You may not need every one for every game situation.
- Troubleshooting should always start with Play_Game.
- The function FC8 can be **very** helpful.
- Do not fiddle with the Command Kernel. It is designed to interface with the M.A.G.E. assembly kernel and has **not** been structured in such a way as to felicitate re-programming. Future command Kernel updates will be

made available as **MajicSoft** develops them.

- Just follow the bouncing ball. New programmers should take it slow and easy. Follow the steps that are laid out above. Working with the code is the best way to fully understand each function and to grasp the complete coding overview. (Remember: Reading is one thing but actually working your way through a problem is probably the best training anyone can provide).



Chapter 3. SCREEN HANDLING

We will explain the different types of screen memory. Scan lines will be discussed and the layout of the Atari's physical memory will be overviewed.

A. Physical Verses Character Screens

The mage system keeps different types of screens in memory. Physical and character screens are two of them. Physical screens are sections of memory arranged so that the Atari hardware recognizes it as a full bit-map graphic screen.

Physical screens can be pointed to as video pages and displayed on screen. Character screens are very different animals indeed. They only represent what is in each Character Cell that the M.A.G.E. thinks is on the screen. The character screen and the MAP are discussed in chapter IV. This chapter will discuss the use of physical screen banks and how the M.A.G.E. uses them.

The M.A.G.E. keeps three copies of the screen in memory (on top of whatever you reserve) for page flipping. MS_Physical% is the address of the start of the screen that you can actually see. MS_Logical% is the address of the screen that gets drawn on (we never draw on the Physical Screen!) and MS_Background% keeps a copy of a fixed background that is copied into the logical screen every page flip. The reason for a background screen is because unlike other systems, the sprites do not keep copies of the background with them. Instead, a screen re-draw is performed. This has the advantage of allowing large numbers of sprites to be drawn very quickly. It has the disadvantage of using slightly more processor time and memory when just a few sprites are displayed.

In addition to these screens, you might want to reserve extra screen banks for your own program's use. This is accomplished via the MS_ScreenInit(N) command. Where N is the number of screens -1 (ie:0=1 screen, 1 = 2 screens, etc..). The extra screens that you reserve with this command also reserve room for the palette, in Degas format. Because of this, we recommend that at least one extra screen be reserved for fading and palette effects. Be warned that this consumes an additional 32,032 bytes of data per screen reserved. Commands, such as MS_FadeIn(), will only work from a reserved screen bank. Screen banks are referred to by number. Their exact addresses can be found via the MS_ScreenAdr() command.

The MS_Vsync20(N,V) is a command you saw in chapter II. This command handles the page flip of the MS_Logical% and MS_Physical% screen addresses and the copying of N groups of 20 scan lines from the MS_Background% screen to the new

MS_Logical% screen. This command is just one of many M.A.G.E. commands used to regulate screen flipping. It is not really necessary to remember which screen is which (the M.A.G.E. keeps the values straight for you), but it is important to note that you should **not** change the values of the screen address variables on your own.

B. Composition of Physical Screen Memory

The Atari ST stores its video in RAM in the form of Scan Lines. Scan Lines in 320 x 200 x 16 color mode are 160 bytes wide. There are 4 bit planes in memory in this mode but they are not stored as you would think. Each word in memory represents 16 pixels on **one plane**. The next word represents the **same** 16 pixels on the next plane. It takes 4 words to complete 16 pixels of data. Then the next 4 words of data handle the next set of 16 pixels. This goes on until 160 bytes (40 words or 320 pixels) are represented. This represents one scan line. If you prefer to shift the screen, you have got to do it on word boundaries because of the way the video is stored.

C. Partial Screen Copies

The M.A.G.E. Vsync commands offer a wide range of copy options, including the option not to copy the background to the logical screen at all. The reason we do this is to offer the programmer the ability to easily implement certain "Tricks" to speed up the operation of the program. By copying only a limited amount of the screen, say 180 scan lines instead of 200, you free the processor to do other things. This allows you to shrink the active page-redraw area and get better game play.

D. Quick Screen Clears

An even better way to get your game to run faster is to use no background screen at all. Instead, use the MS_Vsyns() command and the MS_Clear() command to make for very fast screen re-draws. Games like "Insectroid" and "Evader" use this technique.

E. Screen Masking

Screen masking is the technique that games such as "Xenon II" use for multiple backgrounds. This gives your games depth but eats up a lot of processor time. The M.A.G.E. system allows two basic types of screen (or memory) masking: The full mask or MS_Msk160() is where a sprite type mask is created on the fly. This is by far the slowest method. It does, however, allow you to mask a full set of 16 color scan lines onto a background screen. The MS_Multi8or() routine only OR's your data with the background. This is much faster than the full mask but it creates problems of its own. The OR command only allows certain colors to blend into each other. Basically, you must plan on your color to use two different sets of bits in the color encoding scheme. For example, plan your set background to use bits 0 and 1 and your OR'd background to use bits 2 and 3. Any colors that use bits from both sets can not correctly be OR'd with one another. You could, of course, set up your bit-plane rules however you wish. Just remember that OR'd data should not share bit planes.

F. Displaying Your Screens

It is possible to load in a .PC1 file and display it on the screen with a few simple commands. To load in a .PC1 screen use the MS_DegCLoad() command and the MS_FadeIN() command to show it. If you do not want to fade the screen in, use the MS_MoveM() command to copy the buffer to the MS_Physical% buffer.

G. Storing Your Screens as Inlines

One nice feature of GFA Basic is the INLINE command With this command you can store any data you like in th program, eliminating the need for external files. The M.A.G.E has a full line of features that allow for you to store compresse screen files inside of INLINE statements for later retrieval.

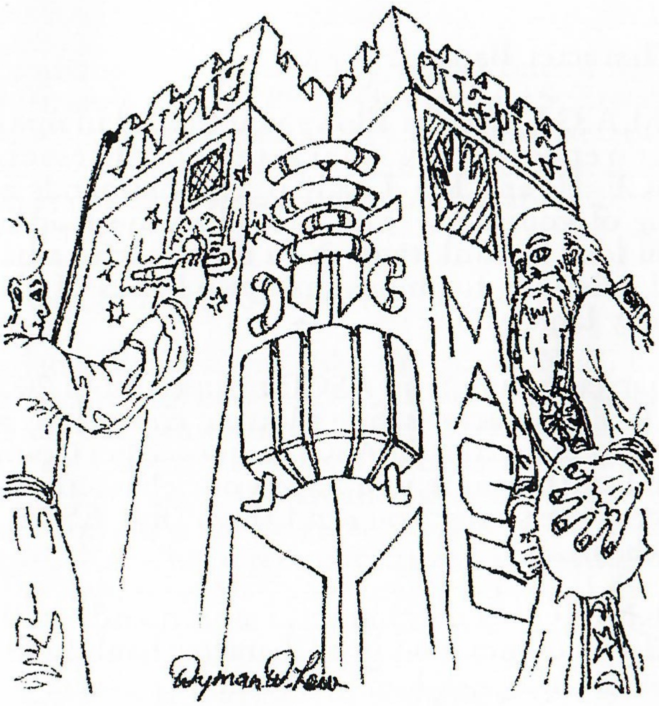
Here is how it works:

- Draw your screen and save it in *.PC1 format.
- Compress the screen using the JamPacker, use ICE/1024/Data/Header & Flash off.
- Write down the length of the file.
- Create an inline in your program of the file length and load in the compressed file.
- When you want to display the screen, simply MS_DeIce() the Inline into the MS_BAckground% buffer, next MS_DegCMem() the file into a screen bank. Use the MS_Copyboth() and MS_fadeIn() commands to show your screen!

H. Scrolling Your Screen

Unfortunately, not all Atari computers allow for smooth scrolling of the screen. The M.A.G.E. system was designed to work on all Atari ST systems so no horizontal fine pixel scrolling is supported. Our Majic Library Update may support such a system if **you** request one! There is a fine system of character based scrolling available in the M.A.G.E. which all computer systems can implement. The MS_Map() commands allow for scrolling and tracking of entire virtual worlds that you can create. See the Character Map Chapter for more info on these commands. Right now, however, we will talk about vertical fine pixel scrolling.

The way to create the vertical screen scrolling effects is to reserve your own memory buffer that is 160 x the number of scan lines your total map is going to be. This is really building a **really** long screen in memory. Now place the variable MS_Background% equal to the start of your buffer. Call the MS_MapDraw() routine and have it draw into the MS_Background%. Add 32,000 bytes to the MS_Background% variable and repeat the MS_MapDraw(). Do this until all of your play field has been created. Now, during game play, you can just change the address of the MS_Background% by plus/minus 160 bytes to scroll your background.



Chapter 4. Character Banks And Maps

A. Purpose of This Chapter

The purpose of this chapter is to familiarize you with the ideas of Character Banks, the Character Screen and the Character Map. It will also familiarize you with some of the most important M.A.G.E. command that utilize these features.

Because of the highly detailed nature of Chapter 10, this chapter will only overview some commands. It will discuss possible implementations of the character command set. If you wish, you might want to skip ahead to chapter 10 and review the character command sub-set. After getting an idea of the types of commands available and their general purpose, you may get more from this chapter.

B. The Character Banks

The M.A.G.E. system allows you to load in up to two .CP1 banks for representing characters on the screen. The MS_LoadCP_1() and MS_LoadCp_2() commands auto-handle the loading of your data. You should always load in bank one before you load in bank two. You may re-load a bank after its initial call, allowing for more than two banks of characters for exceptionally large games.

A Character Bank is an ASCII mapped set of 200 characters. The M.A.G.E. character string plotting commands assume that you have your characters in ASCII sequence (ie: Commands such as MS_CBText() assume you have your character set in order, they convert the string you send them into ASCII values and then plot them).

WARNING: The character commands will not work properly if you do not load in a Character Bank first!! Failure to observe this warning can lead to a computer crash!!

C. Using the Character Banks

To plot a string on the screen you could use a command like this:

```
MS_CBText (0,0,"I LOVE THE M.A.G.E.!",MS_PHYSICAL%)
```

This command tells the M.A.G.E. system to place the string "I LOVE THE M.A.G.E.!" into consecutive character cells starting at location 0,0 and to draw the actual graphics on the physical screen page. The Character Cells are part of an invisible buffer that the M.A.G.E. maintains. This lets you quickly and easily recall what characters have been placed on the screen. What you see on the screen will depend on what you placed into those characters when you created the .CP1 file.

The Command MS_CPeek(0,0) would return the value 73 in the global variable MS_VAR%. This is the ASCII value of the letter "I". By using the character command set (listed and explained fully in Chapter 10) you can Peek, Plot, Poke, Write Text Horizontally and Vertically, Center and fill with your active character bank.

D. What is the Character Screen?

Most video game systems (like the Super NES) build their screens from smaller collections of data called characters. Many video games do the same thing. The M.A.G.E. system is no exception. The Character Screen, which represents what is currently being displayed, is a smaller version of the Character Map (which defines the entire game world). The upshot of all this is that with 200 Character cells (most games use many less) you can build screen upon screen of video graphics for your game.

The M.A.G.E. has two different types of character buffers. The Character Screen is a buffer the size of the screen in cells. To determine what your cell dimensions are refer to the chart in chapter II which shows you the relation of pixel size verse character dimensions. The Character Screen is zero based, thus a 40 x 25 screen has coordinates between 0,0 and 39,24. WARNING: Some M.A.G.E. character commands have no safety net (for speed purposes) and will crash if you try to access outside a legal cell.

The Character Screen will keep a record of all the characters you have plotted on the screen. It will work with the Character Map to allow for the display and handling of larger than the screen backgrounds.

E. Why Have a Character Screen?

The purpose of the Character Screen is to keep a temporary record of the character cells you have plotted on the screen. In order for this to work correctly you must plot the characters on cell boundaries. Please note that some M.A.G.E. commands allow you to plot a character anywhere on the screen, but that this will **not** update the Character Screen. This record is very useful in collision detecting and in setting up nice interface systems for the user.

Without the Character Screen you could not easily tell what background object the sprites were positioned over. In "Thurg N Murg" you can see that the platforms and pickups are all kept in the Character Screen. When the sprites move, they check to see if they hit a platform. They also check to see it they hit a pick-up. The command MS_CPeek() is fast and easy for such purposes. Examine the routines Check_Legal and Collision_Check in "Thurg N Murg" to see exactly how to work with the sprites and Character Screen.

F. The Character Map

The Character Map is simply a collection of character screens. The Character Map provides a permanent record of all the backgrounds used in the game. For instance, when you want to display a level in "Thurg N Murg", you first copy some of the Character Map onto the MS_Background% and Character Screen. This is accomplished by using the command MS_MapDraw(). Now whatever we do to the Character Screen will have no effect on the Character Map (Note: There are exceptions to this!) and if we want our original background back we simply call MS_MapDraw() again.

The Character Map can also be interacted with. The Map Scroll commands can create a "Super Mario World" type effect, allowing you to move around an area bigger than your physical screen. When doing this it is important to remember several things:

- Your Character Map and your Character Screen will have to be updated. If a player takes a pickup from the background you must erase it from **both** areas or you will still have it the next time you draw the map.
- The Character Map must be re-loaded in order to restore any changes you made to it. This is fast and easy if you **inline** your data.
- You can only scroll on character cell boundaries.

G. Loading a Character Map

You can load in a character map by using the command MS_LoadMap(Path$) where Path$ is the path of where your MAP is located. You should always be sure that the character set you have loaded is the same size used to create the original .MAP file or unexpected results may occur.

H. Embedding a Character Map

You can embed a character map in a GFA Basic **inline** statement by following these steps:

- Make an **inline** form of your .MAP file following the instructions outlined in Chapter II.
- Record its final size and then create an **inline** in your program that is the same size.
- Load the file into the **inline** using the HELP and LOAD method.
- Call the M.A.G.E. command MS_MapInl(Adr%) where Adr% is the **inline** you used to store your map data.

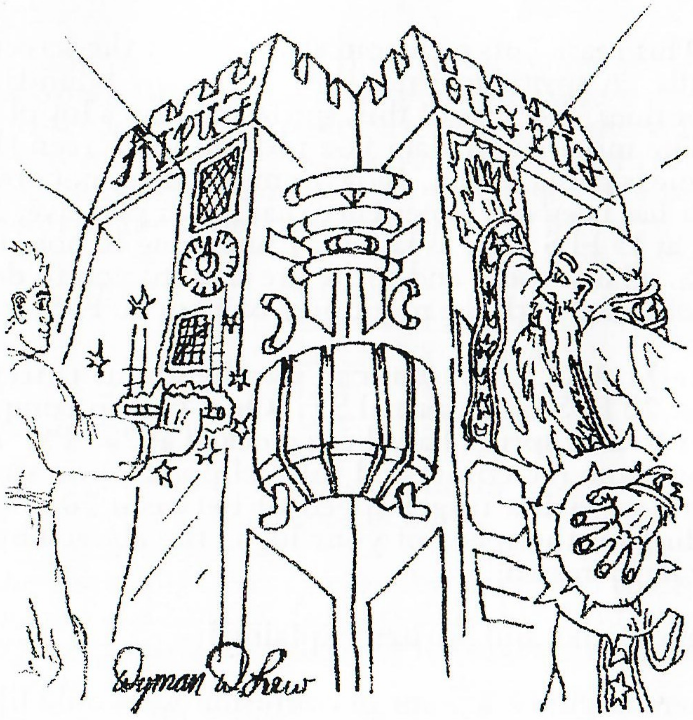
Embedding a Character Map is a very professional method of storing data. It makes your program run smoother in its operation, cuts down on external files and makes it hard for hackers to modify your code.

I. Scrolling With the Character Map

When scrolling horizontally, the M.A.G.E. systems takes advantage of the MS_MAP() functions to do its scrolling. The commands MS_MapUp(), MS_MapDown(), MS_MapLeft() and MS_MapRight() handle the character scrolling for you. Each command will draw a screen of characters based on the X,Y starting coordinates on the Character Map. The Character Screen is also updated along with the MS_MapH& and MS_MapV& variables. To see these commands in action, enter the map editor and press F5 (Scrolling ON). Now draw something on the screen and use the mouse to scroll around. Note: The scrolling in this program works at the fastest rate possible. You can use this speed to judge what type of scrolling speed your game is going to have (about half the map editor's speed is due to sprite movent etc...).

J. Tips for Using Maps:

- Imagine the Character Screen as a "Window" looking into the Character Map. The Character Screen never changes coordinates and locations, while the map can.
- The Character Map should be treated as a permanent record of the background. Only write to it if you plan to reload it the next time you restart your game.
- Use the Character Map command set for horizontal scrolling.
- Take a close look at our demo code for even better examples of how to interface with the Character Screen.



Chapter 5. Sprites

A. Purpose of This Chapter

This chapter is going to introduce you to the M.A.G.E. sprite system. It will explain simple movement and tracking commands. It will also explain offset plotting and some more advanced techniques such as multiple sprite objects and tabled movement.

B. Sprites Verses Speed

Sprites are probably the easiest thing to manipulate in the M.A.G.E. system. A sprite is a graphic image that contains a mask. This mask cuts a custom shape out of the screen that the image fills. A sprite does not show its square boundaries like a character does. Because of this, sprites take up a **lot** of processor time. The more sprite data you plot on the screen the slower your game is going to run. Slow frame rates are not always a bad thing. It has been our experience that a user perceives 25 objects running at 15 FPS to be as fast as a game using 10 objects running at 30FPS. Game speed and sprite use is up to you to decide. Do not get obsessed with the magic goal of 50 or 60 FPS.

The M.A.G.E. system can handle about thirty 16 x 10 sprites at 20 FPS on a normal ST. On a Falcon computer, one hundred 16 x 10 sprites have been clocked at 20 FPS! All sprites are 4 bit-plane (16 color) and fully clipped. You should plan your games with a target speed of between 20 and 25 FPS. Depending on the speed of your logic, the above numbers can actually be increased!!

C. Image Banks and Sprites Explained

There might be a point of confusion we would like to clear up right here and now. And that being what the difference is between sprite images and the actual sprites used during programming. When you are creating an .SP1 file you are **not** creating actual sprites. Instead, you are creating image data that a sprite uses to display itself. An animation file keeps a list of these images in memory so the sprites can be told what sequence of images to use. The declared sprites can use any image or animation they want. For instance, sprite number 0 can take on images 0-199 and animations 1-XX. Sprite number 1 can do the same. You do **not** have to declare a separate sprite for each image created in the M.A.G.E. animation editor.

D. Planning for Your Sprites

You should always sit down and decide how many sprites you are going to need. The first step to programming with the M.A.G.E. is to decide how many sprites will be needed for the player, his shots, the monsters, their shots, bonus sprites, etc... Write this down on a piece of paper. Because of collision checking (discussed in the next chapter), you should always group your sprites together by function. Leave extra slots open if you are not sure how many of a particular type you are going to need. Below is the sprite plan for "Thurg N Murg".

Sprite #	**Function**
0	Player 0's character
1	Player 1's character
2, 3, 4	Player 0's shots
5, 6, 7	Player 1's Shots
8, 9, 10, 11	Bonus numbers
12-20	Monster Sprites
21-30	Monster Shots.

With the above plan you can see that you need to declare 31 actual sprites in the MAGE_Init() command. You now know each sprite's function. This will come in handy throughout your programming session.

E. Loading a Sprite Image Bank

The MS_Loadsp1_1() and MS_Loadsp1_2() commands handle loading of the sprite banks. The first sprite bank must always be loaded first and the second sprite bank must **always** be the same size as the first bank.

Example: MS_loadsp1_1("E:\Thurg\Thurg.SP1")

The above example will load in the sprite bank Thurg.SP1. That is all there is to loading in a sprite bank!!

F. Embedding a Sprite Image Bank

The following steps must be taken to embed a sprite bank. Be warned that you should not do this until the game is completely finished because you must repeat this procedure every time you make a change to the original .SP1 bank.

- Make an **inline** ready file by following the instructions in Chapter II, The Inline Maker.
- Note the final file size and create an **inline** in your program to load it into.
- Using the Help and "L" keys load your data into the **inline**.
- Now use the MS_Sp1Inl_1(Inline_Address%) command to load in the sprite bank.

That is it! Note the MS_Sp1Inl_1() and MS_Sp1Inl_2() commands work the same as the external commands but instead of the path you give the inline name you used to store your sprite bank. The M.A.G.E. will handle de-compressing and installing the images automatically.

G. Loading a GPA File

To load an .GPA file created in the M.A.G.E. Animator use the following command:

MS_LoadGPA(Path$)

This command will automatically load and embed the .GPA file you have created in the M.A.G.E. animation editor.

H. Embedding a GPA File

Here are the steps to embedding a .GPA file. Please note that this should be the last step in your game development due to the fact that you must repeat these steps if any changes have been made.

- Create an inline-ready .GPA file as per chapter II, The Inline Maker.
- Note the file size and create an Inline of the same size.
- Using the Help and L keys, load the data into the inline.
- Use the M.A.G.E. command MS_GPAinl (Inline_Address%) to activate the file.

I. Animating a Sprite

You can tell a sprite to use the image sequence stored in an animation slot by using the command MS_Anim(). This command will not tell the sprite where to be placed. In order to do this use the command MS_Animate() which sets the sprite location as well as its animation sequence. If a single image is desired use the MS_Sprite() command. All these commands activate the sprite (ie: Start drawing it). MS_SpriteClear() is the command you should use to turn a sprite off. Experiment with the above commands to see how these effect a sprite.

To change an animation without activating a sprite, use the MS_AnimO() command. This might be useful when you want to hold a sprite in reserve or do not want to change its status.

The MS_Expode() command allows you to display an animation while causing a collision. Useful if the monster gets killed, but you want him to splatter in some way or another. The MS_Explode() command has a nasty little surprise for the unwary. The global variable, MS_PHit|(), is a flag that the

M.A.G.E. uses for collision checking. If MS_PHit|() for a sprite is **not** zero, a collision check will **not** be performed on the sprite, even if you include it in the collision command parameters. Make sure to call the MS_SpriteClear() command after you have used MS_Explode() to avoid any collision checking errors. If a sprite does not seem to be obeying a collision check then this should be one of the first things to check for errors.

J. Moving a Sprite

A sprite can be moved in a variety of ways in the M.A.G.E. system. Direction, Tracking, Patterns, Offsets, OffsetFs and user controlled movement, will be discussed.

K. Direction Movement

A sprite can be told to move in a direction. The command MS_Direction() command will move a sprite a certain number of steps every game loop for a certain number of game loops and then turn the sprite off. The MS_LDirection() command will perform the same function but will **not** turn the sprite off.

The sprite will be turned off if it exceeds the boundaries you have set up in Init_Game.

L. Tracking Movement

You can assign a target sprite for the sprite to track. The command MS_Track() allows you to designate a target, speed and length to track. The sprite tracking will turn off once the length reaches zero. The track command does not effect animation.

M. Patterns

The MS_Pattern() command tells a sprite to follow a fixed path and then turn off. The MS_LPattern command resets the sprite to the beginning of the pattern instead of turning it off.

Patterns are a very fast way to move things. Games such as "Xenon II" use patterns to move their aliens.

N. Offsets

Offsets work just like patterns except they are not in a fixed path. Instead, they are fixed to a path based on their starting location. In other words, the offset will always follow the same sequence of steps (left, right, right, etc...). But since you can start them anywhere on the screen, they have different screen coordinates than the original. Games like "Insectroid" and "Megapede" use this type of command. This type of movement is slightly slower than the pattern method.

O. OffsetF

OffsetF's are offsets that continually change based on the sprite they are attached to. An example would be a circle OffsetF. You launch a sprite using this OffsetF based on your main character. Now, the sprite following the OffsetF will circle your main character no matter where it moves.

This type of movement is the slowest, yet most powerful form of movement the M.A.G.E. has to offer.

P. Global Movement

Used in games like "Defender" and "Gauntlet", this type of movement shifts a number of sprites a certain number of pixels. The command MS_MoveSprites() will do this for you. Excellent when using scrolling backgrounds or if you want to move a group of sprites together.

Q. User Controlled Movement

Sometimes the M.A.G.E. movement commands are not enough. This is where the global variables MS_SHor&() and MS_SVer&() come in. These arrays hold the horizontal and vertical position of every sprite you have declared. You can change these values from within the program in order to move your sprite. "Thurg N Murg" uses this method to introduce gravity into its jumps and falls. The actual use and manipulation of these variables will depend on the type of game you are programming. Study the enclosed source code for a more detailed explanation on how to manipulate these variables.

R. Other Commands for Sprites

Plotting a Sprite:

A sprite image can be plotted on the screen. If you want to "Stamp" something onto the screen this is the way to do it. The command MS_SPlot() will perform the task.

S. Multiple Sprite Images

In some cases you may want to display an image on the screen that is bigger than a single sprite image cell. This is easily accomplished through two different methods of plotting.

The first method is the multiple sprite method. This is done by simply storing the image in separate sprite image cells. Then all you need to do is use several sprites to display the image. A 2 sprite high by 2 sprite wide image can be displayed by using 4 sprites. Another way to keep the sprite together is to move a single sprite as per the methods above and then updating the MS_SHor&() and MS_SVer&() variables of the other sprites that make up the image. For instance, MS_SHor&(1) = MS_SHor&(0)+16. Finally, you can move the images using the MS_MoveSprites() command.

The second method is to use the MS_SpriteBlock() command. This requires that you draw each frame of the large image on a screen and have loaded it into a screen bank somewhere. You should not use this method unless you are manipulating rather large pieces of data. Use the screen banks and the command MS_ScreenAdr() to ease the handling of your sprite blocks.

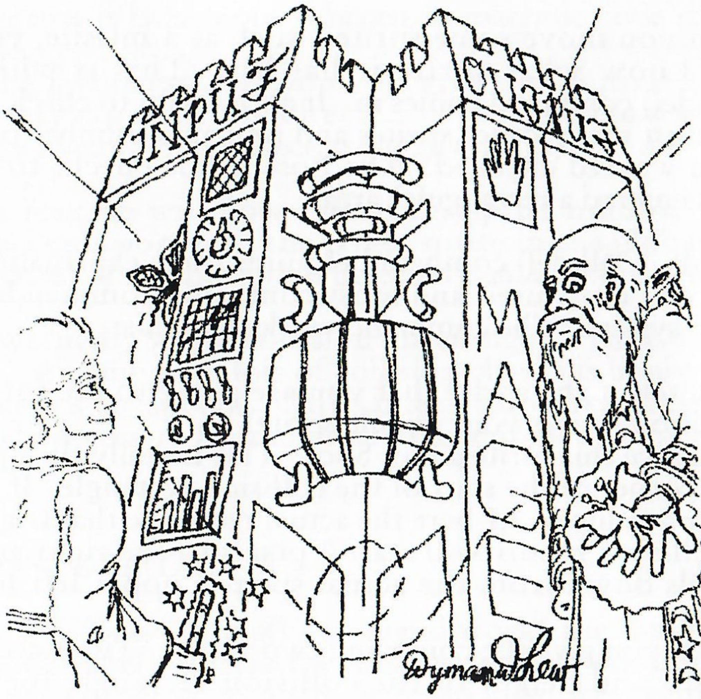
In any case, moving large blocks of sprite data is sure to slow your program down. Keep activity down when a big sprite is on the screen by limiting the number of other sprites that appear at such times.

T. Tips for Using Sprites

- Plan your image data so that the fewest sprites possible can be on the screen at any one time.
- Large images can be displayed via the MS_SpriteBlock() command.
- Movement takes on many forms in the M.A.G.E. Experiment with each form so that you know which one is the right one for you.
- Review Chapter 10 to get more information on the sprite commands.

MajicSoft®

"We Put The Majic In The Software!"



Chapter 6. Collision Checking

A. Purpose of the Chapter

This chapter will discuss the two main types of collision checking. These are sprite collision checking and character collision checking. We will also discuss sprite organization for the best results during collision checking. Zone Checking will also be discussed.

B. Sprite Collision Checking

When you move some sprites, such as a missile, you will want to know what sprite it has hit. This is where the MS_Collide() command comes in. It allows you to check a single sprite against many other sprites and returns a number based on what you wanted checked. The Bosscollide checks to see if a sprite has entered a rectangular area.

The MS_Collide() command requires some explanation, as it is one of the most used and most confusing commands in the M.A.G.E. system. The command works like this:

- You tell it the sprite that you are going to use for the collision (this is your missile sprite).
- You give this sprite a Hot Spot. This is really the upper left corner of the start of the collision rectangle. If you place a value of 2,2 here the actual rectangle that triggers a collision report will start 2 pixels to the right and 2 pixels down from the actual sprite's upper left hand corner.
- Next, you give the command two values. These are the width and height of the collision rectangle for the missile sprite.
- Now the command expects the same type of information that it will apply to the other sprites (the Target sprites). This is in the same format as the above information. The reason to have two collision rectangles is very simple. Your missile may be much smaller than the monsters. Thus, you would want to make the collision rectangle around your missile to be pretty small (**not** the size of the whole sprite image). You **would** want the monsters to have a pretty big trigger area. This is why we have a second trigger rectangle. The MS_Collide() command simply compares all these rectangles and reports the first one that overlaps.
- The command now wants to see the **highest** numbered

sprite to check. This command checks downward because it is faster to subtract in machine code than to add.

- The last parameter you will give the routine is how many sprites to check. This routine works downward. So to check from 20 to 12, give it a 9.

The routine will now return the sprite number that first triggers a collision with the target sprite in the global variable MS_VAR%. If no collision is detected then a zero is returned.

Be warned! Sprite collision checking is a time consuming process. A game with lots of collision checks is likely to drop a few Vsyncs in speed. Only check those sprites that are absolutely necessary for the game to function!

C. BOSSCOLLIDE

The MS_BossCollide() command works the same way that the MS_Collide() function works. Instead of checking sprites against each other, it compares a group of sprites against a fixed rectangle. This is used when you want to see if your missile sprites have run into a large section of the screen, possibly a large Boss alien of some type.

D. Character Collision Checking

There are several ways to check for sprite collisions and the Character Screen. We will cover them in the following descriptions. Character Screen checking is most often used in looking into the background for walls, platforms and pickups. The Character Map should not be used for collision checking because of the special problems that this poses.

CBitPeek()

Use this command to check under the sprite. You need to set the X,Y coordinates to the middle of the sprite if you want the correct character cell. Example (Using 16 x 10 sprite size):

```
X&=MS_Shor&(0)
Y&=MS_Sver&(0)
add X&, 8
add y&, 5
CBitPeek (X&, Y&)
```

CPeek

Use this command when you know what cell you are over. If you want to do a manual calculation of the character cell for accuracy or some internal factor, use this command. Here is an example of a sprite that you want to check just ahead of (16 x 10 sprites):

```
X&=MS_Shor&(0) ! Get the X,Y of sprite 0.
Y&=MS_Sver&(0)
Add X&, 17 ! Look Right one cell.
Add Y&, 5
Div X&, 16 ! This is the size of the sprite!
Div Y&, 10
CPeek (X&, Y&) ! Look at this cell.
```

Both of these commands return the value of the cell in the global variable MS_Var%.

You may also use the same technique to look into the Character Map. This would be useful if your sprite went off the physical screen but still had to worry about hitting something in the background.

Zone Checking

Zone checking was set up to provide an easy way to check boxes. This command set is more about building user interfaces than about sprite collisions. Their primary use is in building user interfaces. They can quickly check through up to 255 rectangles and return which one an X,Y coordinate is in.

The Zone commands are listed in Chapter 10. There are only a few things that need to be said about them here.

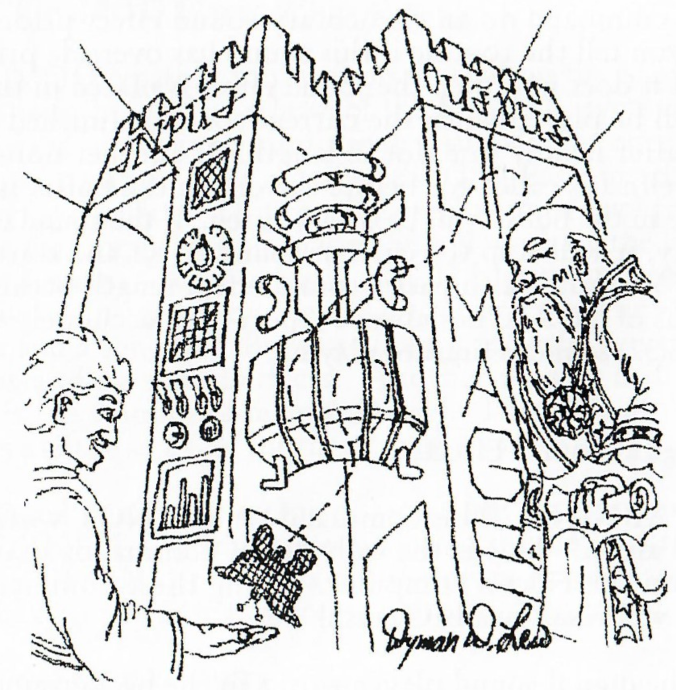
The Zone commands will only return the first rectangle violated. Be careful of how you set them up so that you do not fail to detect multiple zone violations. You also need to be aware that this command could be used in establishing areas in a video game. We allow them to be moved around for just that purpose. Walls could easily be bit-mapped and defined with zone rectangles instead of using character cells to build them.

E. Tips for Collision Checks:

- Keep your sprites grouped so that one collision check command can handle the whole set of sprites.
- Keep collision checking down to a minimum. Lots of collision checking can slow down a game in a hurry!
- Use Sprite collisions to compare sprites.
- Use Character collisions to check the background.
- Use Zone collisions to help with bit-mapped backgrounds and button-like user interfaces.
- Remember that the MS_Collide() command works in a **downward** fashion!!

MajicSoft®

"We Put The Majic In The Software!"



Chapter 7. Sound Effects

A. Purpose of this Chapter

The purpose of this chapter is to explain the three different ways the M.A.G.E. system produces sound effects. We will discuss the built in sound effects, the interrupt driven digital sound commands and the chip music .MOD interface system.

B. Built in Sound Effects

The M.A.G.E. system's MS_Sound() command triggers one of the 101 built in sounds to be played. Sound 100 is a null sound and is used to turn off the existing sound effect early. A sound number from 0 to 100 is used. The other parameters used in this command do an elementary sound effect priority sort. First, you tell the routine if this sound has override priority or not. If it does not, then the sound effect is placed in the buffer and will be played when the current sound is finished playing. The buffer is only one slot in length. If another non-priority sound effect is called for before the one in the buffer is played, the one in the buffer will be over written. If the sound effect has priority, it will stop the current sound effect and start to play itself. You must also pass a sound effect length because early versions of TOS do not always report back accurately when an XBIOS(32) sound is finished playing.

C. Digital Sound Effects

WARNING: This command set will **NOT** work on the Atari Falcon. This is the only set of commands that do not work on the Falcon computer. Using these commands on a Falcon will result in a **BIG** crash!

The digital sound player works in the background. Your program will continue to operate when sound effects are being played with these commands. The MS_SamplePlay() command begins playing a sample. The MS_SampleStop() command turns off a sample. The MS_SampleWait() command pauses your program while the sample is playing.

The digital sound player uses ST-Replay format sampled sounds and can handle speeds of 3Khz to 15Khz. Lower speeds may result in a computer crash. The faster the playback speed is, the more time the sample replay routine requires from the processor. Most real time sound effects in the M.A.G.E. system should stay near the 3Khz range.

It is best to use **inline** statements to embed your sampled sound effects directly into your program.

D. The Chip .MOD Player (Chip Music)

The .MOD creation tool was discovered by the MajicSoft staff just weeks before the release of the M.A.G.E. system. We were delighted with the possibilities that this system opened up for game programmers. We modified the GFA Basic interface to use the replay routine. This routine works in the background but unlike the sampled sound routine, it **will** work on the Falcon computer system. These types of .MOD files do **not** contain digital music. Instead they contain a sophisticated chip music definition and a song file. By using these creatively you can get some outstanding sound effects. Another plus to using this interface is the amount of memory saved. This type of .MOD music uses as little as 1% of the memory that digital samples use.

The Mod interface consists of a creation program named, "MUSICMON.PRG", which can be found on the Extra Disk. The "MUSICMON" program is **not** the property of MajicSoft. It is provided to you as shareware, so please register your copy.

E. Using the MOD Player

The mod player interface has several commands to ease the construction of multi-voice sound effects. The first thing you need to do if you intend to use the MajicMod interface is to call the routine MS_Init_Player(). Pass it the total number of MODs (zero based) that you are going to use throughout your game. After you have done this, you need to call MS_Embed Mod for each MOD that you want to use during the game. It is important to do this before you use the sound so the replay routine does not crash. The following are the steps to follow:

1) Make your MODs and note their length.
2) Load them into **inline** statements.
3) Call MS_Init_Player().
4) Embed the Mods using MS_Embed_Mod().
5) Play them using MS_Play_Mod().

F. MODS and Sound Effects

If you create a MOD that is only a couple of notes long you can replay this "Song" as a custom sound effect. This is the real power of this interface. Another good technique is to make the pattern very short so it takes up even less memory and clears out nicely.

Chapter 8. Miscellaneous Commands

A. Purpose of this Chapter

This chapter will discuss commands and techniques that do not easily fit into the other chapters.

B. MS_BLIT Inline

The MS_Blit.INL kernel is only used when you want extra speed in your Blit routines (ie: Moves and Screen Masking). It takes up an additional 30K plus uses a buffer of 32K. It does add a real speed improvement to your code. Use this when you are programming for a 1 megabyte machine or higher.

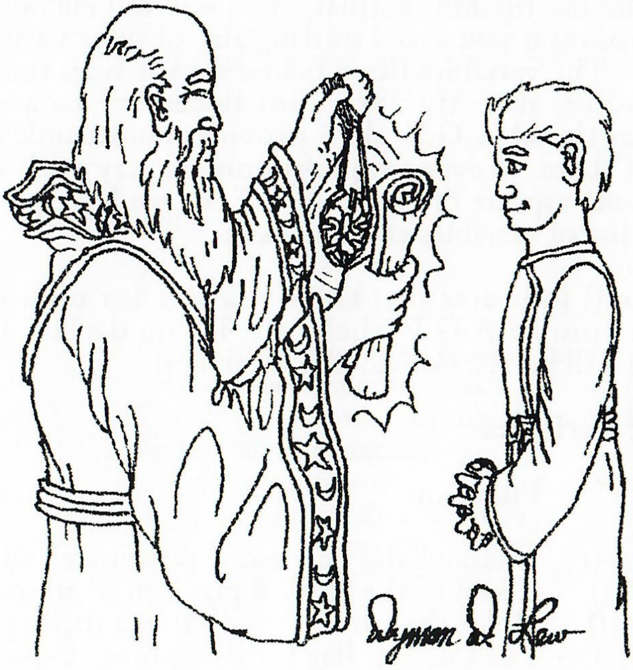
C. Star Plotting

The star plotting commands (as seen in chapter 10) are used to create a variety of effects. The multi-colored star field and snow effects are the most common. We need to note a few oddities of these routines so you do not become confused. First of all, the stars will move in the direction **opposite** of the joystick direction you give the routine. This is because the routine is designed to simulate two dimension backgrounds. Another quirk not apparent is that the H and V directions are simply magnitudes. They may **not** contain a negative value.

Remember to Vsync and re-draw or clear your screen before plotting your stars. Stars do not self-erase.

D. MS_16/32 Inlines

It may not be apparent to some using this code but you will need to load in one of these inlines into your shell before starting your programming. You load in the inline based on which size character/sprite you will be using; 16 pixel wide or 32 pixel wide. You can not mix the two sizes.



Chapter 9. Important Variables

A. Purpose of this Chapter

Congratulations to you for completing the major portion of this tomb. This chapter will discuss the important internal variables that you can use in your programs. These are advanced features of the M.A.G.E. and it is not necessary to know these variables in order to use the M.A.G.E..

B. Using M.A.G.E. Variables

The M.A.G.E. system has a collection of internal variables. All variables use the MS_ format. You will not encounter global data corruption if you avoid starting any of your variable names with MS_. The variables listed below are the ones that you may want to access directly. It is not necessary to access these variables as the M.A.G.E. shell provides commands that work with all of them. Advanced programmers may wish to directly control some aspects of the M.A.G.E. so we have provided the following list of variables and functions.

Note: The () indicates that there is a slot for each sprite you declare in your M.A.G.E. shell. (ie: If you declare 100 sprites then there will be 100 elements in all arrays)

C. Sprite Variables

Variable	Function
MS_SHor&()	- Contains the horizontal position of all sprites.
MS_SVer&()	- Contains the vertical position of all sprites.
MS_Sprite|()	- Holds the current image that a sprite is using.
MS_SFlag|()	- The On/Off flag for the sprites. 0=Sprite is off.
MS_PHit|()	- PreventHit Flag. 0=Sprite is allowed to be collision checked. Anything else will mask the sprite during any collision checking commands.

D. Animation Variables

Variable	Function
MS_AniFCnt|()	- Counter that points to the current animation frame the sprite is using.
MS_AniMode|()	- What style of animation is taking place.
MS_Anims|()	- Which animation sequence the sprite is using.

MS_AnimFlag&()	- This holds the type of animation playback to use: 0 = Repeat over and over. 1 = Play animation once and turn sprite off. 2 = Play animation once and leave sprite on.

E. Movement Variables

Variable	Function
MS_DirH&()	- Pixels to move horizontally during sprite direction commands.
MS_DirV&()	- Pixels to move vertically during sprite direction commands.
MS_Dirl|()	- Length to move during direction commands in game cycles. 255 = Always.
MS_DMove|()	- The direction the sprite is moving during the track commands. Based on the GFA Basic STICK table.
MS_MoveH|()	- Pixels to move horizontally during sprite tracking commands.
MS_MoveL|()	- How many cycles to track. 255 = Always.
MS_MoveV|()	- Pixels to move vertically during sprite tracking commands.
MS_PatmS&()	- Holds which pattern the sprite should use.
MS_PatmC&()	- Counter to hold which step of the pattern to use next.
MS_OffmS&()	- Holds which offset pattern the sprite is using.
MS_OffmC&()	- Counter to hold which step of the offset to use next.
MS_Track%()	- Holds the sprite that this sprite is tracking.

F. Star Plotting Variables

Variable	Function
MS_Stars&	- Total number of stars to plot.
MS_StarH1&	- Left horizontal boundary for stars (0-319).
MS_StarV1&	- Upper vertical boundary for stars (0-199).
MS_StarH2&	- Right horizontal boundary for stars (0-319).
MS_StarV2&	- Lower vertical boundary for stars (0-199).
MS_StarC&()	- Color of the star.
MS_StarH&()	- Horizontal position of star.
MS_StarPH&()	- Pixels to move horizontally (0-32000).
MS_StarPV&()	- Pixels to move vertically (0-32000).
MS_StarV&()	- Vertical location of star.

G. Screen Variables

Variable	Function
MS_Background%	- Points to the beginning of the background screen.
MS_Logical%	- Points to the beginning of the logical screen.
MS_Physical%	- Points to the beginning of the physical screen.
MS_Buffer%	- Points to a 32K buffer ares (if using MS_Blits).

H. Miscellaneous Variables

Variable	Function
MS_Var%	- Always returns the value of a function.
MS_MapH&	- Map Horizontal pointer location.
MS_MapV&	- Map Vertical pointer location.

I. Global Variables

All of these varibles have different functions according to when they are used in the M.A.G.E. shell. You may freely use these global varibles if you keep in mind that they can be corrupted at **any** time.

MS_File$
MS1$, MS2$, MS3$
MS1%-MS20%
MS1&-MS20&
MS1|-MS20|
MSH&, MSV&

J. Additional Character Variables

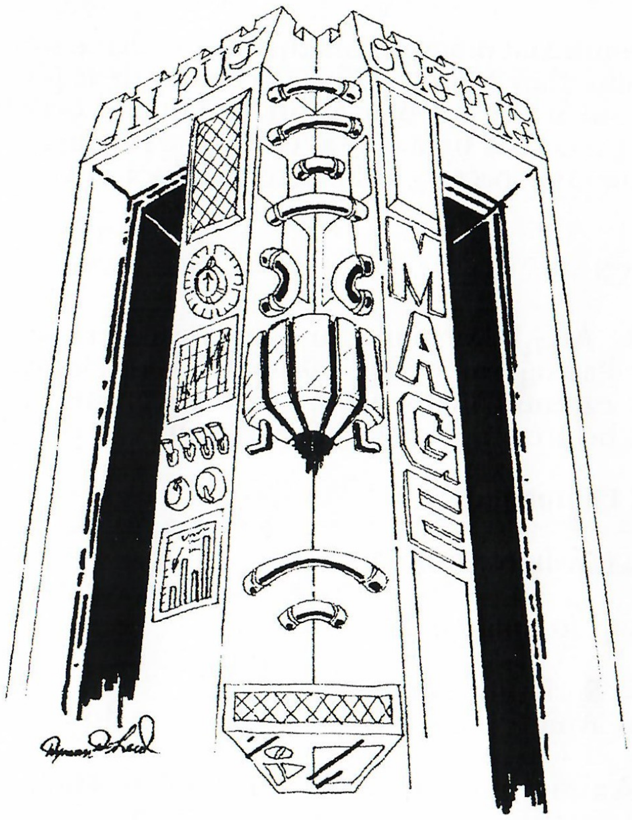
Variable	**Function**
MS_Cp1%	- Address of start of character data.
MS_Vc&	- Vertical pixels of a character.
MS_Cb&	- Bytes per character.
MS_Cm&	- Character Mode (0-16) 1= 32 Pixels wide.
MS_Ch&	- How many characters will fit horizontally on the screen?
MS_HC&	- Horizontal pixels per character.
MS_Cv&	- How many characters will fit vertically on the screen.
MS_CBuf1%()	- Holds the character image data in bank 1.
MS_CBuf2%()	- Holds the character image data in bank 2.

K. Additional Sprite Variables

Variable	Function
MS_Sp1%	- Holds the location of the sprite image data.
MS_SpriteDump&	- Tells the program to dump sprite data to printer.
MS_Sb&	- Bytes per sprite image.
MS_Sm&	- Sprite mode (0-16) 1=32 pixel wide sprites.
MS_SBuf1%()	- Holds the sprite images for bank 1.
MS_Sbuf2%()	- Holds the sprite images for bank 2.

L. Additional Effects Variables

Variable	Function
MS_FlashC%	- Address of flash colors.
MS_FlashOn&	- Flag for turning flash on/off.
MS_Cycle&	- Color Cycle flag.
MS_Vs1%	- Used in Vsync counters.
MS_Vs2%	- Used in Vsync counters.
MS_VsCnt%	- How many Vsyncs to delay.
MS_VsCnt&	- Used to count Vsyncs.
MS_ProgramRun&	- Used to flag a seperate program run upon exit.
MS_SingleStep&	- Flag to trigger single step mode.
MS_Screens%()	- Holds the 32K screen banks.
MS_EffectScr%()	- Array that holds source for effect array.
MS_EffectDst%()	- Array that holds detination for effect array.
MS_Buff%()	- Holds Buffer Data.



Chapter 10. Command Summary

A. Purpose of this Chapter

The purpose of this chapter is to give a detailed description of every command in the M.A.G.E. library. If you read no other chapter this is the one to use. The commands are arranged in alphabetical order.

B. Notation:

The command descriptions that follow have some helpful notation rules they follow. When a coordinate is given in X&, Y& locations it is a bit map value (0-319 and 0-199). If the variable is proceeded by a "C" as in cX&, cY& then you know the command is expecting a character cell location and not a bit map value.

WARNING!

The M.A.G.E. system commands use little or no error checking. Passing the wrong values will likely cause a system crash!! Be careful when working with uncertain values. Save your work before proceeding with a test run.

C. List of Commands

MS_Anim (Sprite&, Anim&)

Sets a sprite animation.

Sprite&	= Sprite affected
Anim&	= Animation slot to use.

Notes: Anim& must be a value from 1 to the number of animations created in your .GPA file.

MS_Animate (Sprite&, Anim&, X&, Y&)

Sets a sprite location and animation at the same time. Frequently used to launch a sprite. This command turns the sprite on.

Sprite&	= Sprite affected.
Anim&	= Animation slot to use.
X&, Y&	= Position of sprite.

MS_Boffset (Sprite&, Offset&)

Sets a sprite to move on an offset pattern with boundary controls. This means that the sprite will de-activate if it exceeds the screen boundaries. This command turns the sprite on.

Sprite&	= Sprite affected.
Offset&	= Offset to use.

MS_BossCollide (X&,Y&, HspotH&, HspotV&, TriggerH&, TriggerV&, SprHSpot&, SprVspot&, SprHtrigger&, SprVtrigger&, Start&, Amount&)

Used to test if sprites have moved into a set location. Usually used to check collision with a graphic image made up of several sprites or an image cut and pasted from a separate screen instead of a sprite bank.

X&, Y&

Upper left of image.

HSpotH&, HSpoitV&

Hotspot to use for this image (added to X&,Y&).

TriggerH&, TriggerV&

Added to hotspot to create the rectangle that triggers a collision report.

SprHSpot&, SprVSpot&

Hot spot offset for sprites to use.

SprHTrigger&
SprVTrigger&

Trigger distance for sprites to use.

Start&

Starting sprite to check. This should be the **highest** number sprite that you want to collision check.

Amount&

The number of sprites **down** that the routine will check. The answer is returned in the global sprite MS_Var%. If no collision has occurred then a zero is returned. If a collision has been detected, then MS_Var% will return the value of **amount** when it occurred. This value may have to be modified to return the correct sprite number. For instance, you checked starting at sprite number 20 and checked five sprites. The number you would get would be from 1 to 5. In order to get the correct sprite number the following formula would work: Sprite& = MS_Var%+15.

MS_Cbbtext (X&, Y&, String$, Screen%)

This command allows you to place character text at any bit map location. This command does **not** update the Character Screen.

X&, Y&	= Starting position of string.
String$	= String to plot.
Screen%	= Starting address of screen you want to draw on.

MS_Cbitpeek (cX&, cY&)

This command peeks into the Character Screen at cell X&, Y& and returns the value of the cell in the global variable MS_Var%.

cX&, cY& = Cell position to PEEK.

MS_Cbitpoke (X&, Y&, CHAR&, Screen%)

Converts a bit map location into the nearest character cell location and plots the character Char& on the screen and pokes its value into the Character Screen. Character will be draw on the screen at the cell coordinates.

X&, Y&	= Bit map position of the character.
Char&	= Character to plot.
Screen%	= Starting address of screen to draw character on.

MS_Cbplot (cX&, cY&, Char&, Screen%)

Plots an image at cell X&, Y& but does not update the Character Screen.

cX&,cY&	= Character cell location.
Char&	= Character to plot.
Screen%	= Starting address of screen to draw the character on.

MS_Cbtext (cX&, cY&, String$, Screen%)

Draws a string on the screen but does not update the character screen.

cX&,cY&	= Character cell to start plot on.
String$	= String to plot.
Screen%	= Screen address to draw images on.

MS_Center (cY&, String$, Screen%)

Draws a centered string on the screen and updates the Character Screen.

cY&	= Vertical cell of string.
String$	= String to plot.
Screen%	= Screen address to draw the images on.

MS_Charbank (Bank&)

Allows bank switching in the M.A.G.E. system.

Bank&	= Bank to make active (1 or 2).

MS_Clear (Screen%)

Clears a complete 32K screen. Faster than GFA CLS command.

Screen%	= Screen starting address.

MS_Clear2 (Screen%, Lines&)

Clears memory in increments of 2 scan lines (320 bytes) at a time.

Screen%	= Screen starting address.
Lines&	= How many groups of 2 scan lines to clear.

MS_Clear20 (Screen%, Lines&)

Clears memory in increments of 20 scan lines. This command works **downward**, in other words it learns the bottom of the screen first.

Screen% = Address of bottom of memory you want to clear.
Lines& = How many groups of 20 scan lines you want to clear.

MS_Clearkey

Waits until keyboard buffer is clear.

MS_ClearStick (Joystick&)

Waits until the joystick button is FALSE.

Joystick& = Stick to check (0-1).

MS_Clickoff

Turns the keyboard click off.

MS_Clickon

Turns the keyboard click on.

MS_Cmbitpoke (X&, Y&, Char&, Screen%)

Converts bit map coordinates into character cell location. It then plots an image and updates **both** the Character Screen and the Character Map.

X&, Y&	= Bit map location of plot.
Char&	= Character to plot.
Screen%	= Starting address of screen to draw image on.

MS_Cmplot (cX&, cY&, Char&, Screen%)

Plots a character on the screen and updates both the Character Screen and the Character Map.

cX&,cY&	= Character cell location to plot.
Char&	= Character to plot.
Screen%	= Starting address of screen to draw image on.

MS_Cmtext (cX&, cY&, String$, Screen%)

Places an entire string of characters onto a screen and updates both the Character Screen and the Character Map.

cX&,cY&	= Starting character cell location of string
String$	= String to plot on the screen.
Screen%	= Starting address to plot the images on.

MS_Collide (Sprite&, HSpot&, VSpot&, HTrigger&, VTrigger&, SprHSpot&, SprVSpot&, SprHTrigger&, SprTrigger&, Start&, Amount&)

Used to do collision checking between sprites. Compares the location of a single sprite to that of a group of sprites and returns the sprite number of the first collision detected in the global variable MS_Var%. See MS_Bosscollide for an example.

Sprite&

Sprite to check against.

HSpot&, VSpot&

Hot spot offset for Sprite&.

HTrigger&, VTrigger&

Horizontal and Vertical distance from hot spot to create a trigger rectangle.

SprHSpot&, SprVspot&

Hot Spot offset to use on the rest of the sprites.

SprHTrigger&, SprVTrigger&

Horizontal and Vertical distance to add to the SprHspot values to form a collision trigger rectangle.

Start&

Highest sprite number in the group to check.

Amount&
How many sprites to check. Works **downward.**

MS_Colorcycle (Switch&, Start&, End&, Speed&)

Used to perform color cycling. The color cycle is shifted once per game loop and is **not** interrupt driven.

Switch&	= Controls direction and activity of color cycling. 0: Off, 1: Right, 2: Left.
Start&	= Starting color register to cycle.
End&	= Ending color register to cycle.
Speed&	= How many game loops to skip between cycles.

MS_Convert (X&, Y&)

Converts bit map coordinates into character cell coordinates. The Cell location is returned via the global variables MSH&, MSV&.

X&,Y&	= Coordinates to convert.

MS_Copy_Logical_To_Physical (Start1&, Start2&, Lines&)

Copies a section of the Logical screen to the physical screen.

Start1&	= Starting scan line of the Logical screen.
Start2&	= Starting scan line of the Physical screen.
Lines&	= How many scan lines to copy.

MS_Copy_Logical_To_Background(Start1&,Start2&,Lines&)

Copies a section of the logical screen to the Background screen.

Start1&	= Starting scan line of the Logical screen.
Start2&	= Starting scan line of the Background screen.
Lines&	= How many scan lines to copy.

MS_Copy_Background_To_Logical(Start1&,Start2&,Lines&)

Copies a section of the Background screen to the Logical Screen.

Start1&	= Starting scan line of the Background screen.
Start2&	= Starting scan line of the Logical screen.
Lines&	= How many scan lines to copy.

MS_Copyto (Source&, Destination&)

Copies the Source screen bank to the Destination screen bank.

Source&	= Screen bank number to copy from.
Destination&	= Screen bank number to copy to.

MS_Cpeek (cX&, cY&)

Peeks the Character Screen and returns the value of the cell in the global variable MS_Var%.

cX&, cY&	= Character cell location to PEEK.

MS_Cplot (cX&, cY&, Char&, Screen%)

Plots a character onto the screen and updates the character screen.

cx&,cy&	= Character cell location to plot.
Char&	= Character number to plot.
Screen%	= Screen address of the screen to draw the image on.

MS_Cscreen (Char&, Screen%)

Fill the screen and the Character Screen with character Char&.

Char&	= Character to fill with.
Screen%	= Screen address to fill with the images.

MS_Ctext (cX&, cY&, String$, Screen%)

Prints a string on the screen and updates the Character Screen.

cX&,cY&	= Character cell to start plotting at.
String$	= String to plot.
Screen%	= Screen address to draw the actual images on.

MS_Cvtext (cX&, cY&,String$, Screen%)

Prints a string vertically on the screen and updates the Character Screen.

cX&,cY&	= Character cell to start plotting at.
String$	= String to plot.
Screen%	= Screen address to draw the actual images on.

MS_Degcload (Bank&, Path$)

Loads a compressed DEGAS format(.PC1) picture into a screen bank.

Bank&	= Screen bank to load into.
Path$	= Filename to load.

MS_Degcmem (Bank&, Adr%)

Decompresses a compressed DEGAS format (.PC1) into a screen bank.

Bank&	=	Bank to decompress into.
Adr%	=	Starting location of the .PC1 file in memory.

MS_DeIce (Source%, Destination%)

Decompresses ICE format compressed data into a destination memory block. The following ICE options are strongly recommended: Compress ratio at 1024, Flash and Header OFF.

Source%	=	Memory location of the compressed data (usually an INLINE).
Destination%	=	Start of memory block to decompress into.

MS_Direction (Sprite&, Hspeed&, VSpeed&, Length|)

Moves a sprite in a set direction for Length& steps and then turns it off.

Sprite&	=	Sprite to move.
HSpeed	=	Horizontal pixels to move per step.
VSpeed&	=	Vertical Speed to move per step.
Length|	=	Number of steps to move in this direction. 255 = move until sprite exceeds a screen boundary.

MS_Drawsprites (Screen%)

Draws all active sprites on the screen address screen%. Usually not called by the user.

Screen%	=	Screen address to draw the sprite images on.

MS_Effect160 (Source%, Destination%)

This routine processes an M.A.G.E. effect array (see MS_Effectinit() & MS_Effect160set()) between the source and destination addresses.

Source%	=	Address of source graphic data.
Destination%	=	Address of destination of graphic data (usually a screen address).

MS_Effect169init (Amount&)

Initializes the ME_Effect arrays so they can be set by the MS_Effect160set() command.

Amount&	=	How many effect lines are you going to use.

MS_Effect160set (Index&, Source&, Destination&)

This sets up the powerful MS_Effect160() command. What the effect command does is process a list that tells it how to copy from the source memory block to the destination block.

Index&	= Which entry (starting at 0).
Source&	= How many scan lines to subtract from the source address.
Destination&	= How many scan lines to add to the destination address.

Source& and Destination&	= Can be negative to reverse direction of the copy.

This command can create some powerful effects. The most famous is the barrel scroll effect, which is the effect of a rolled tube of images scrolling by. To achieve this effect you need to think of the tube in scan lines. The top and bottom of the tube compress the image while the center shows the entire image. In order to achieve this effect quickly, we do not really compress the data at all but keep a normal screen chunk of memory that we copy from selectively. By skipping scan lines on the top and bottom of our copy we achieve a compressed look and by copying straight across the image we get a non-compressed look.

A typical structure for a tube is (in scan lines skipped):
2,2,2,2,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,2,2,2,2

In the above example, the effect routine would skip 2 scan lines and then copy the third, add 2 then copy, etc.. until it got to the last piece of data.

The typical source of the image data is in a screen bank and the destination is usually MS_Logical%. By moving the source address around we can achieve a scrolling effect.

MS_Explode (Sprite&, Anim&, Length&)

Causes a sprite to halt all movement and display an animation. The sprite will be masked out of all collisions for the duration of the animation.

Sprite&	=	Sprite to effect.
Anim&	=	Animation to use.
Length&	=	Length of explosion (animation will cycle if Length& > Anim&)

WARNING: The Variable MS_Phit|(Sprite&) is set to >0 in this command. If you re-launch the sprite before the explosion is over the MS_Phit|() will **not** be reset. This will mask your sprite from collision checks until you reset the MS_Phit|() flag to zero! Be careful with this.

MS_Fadein (Bank&, Flag&, Speed&, Start&, End&)

Fades in a screen bank.

Bank&	=	Screen bank to fade in.
Flag&	=	0: Fade Palette only.
		1: Fade in Palette and Screen data.
Speed&	=	Speed (in Vsyncs) to fade in (0=fastest).
Start&	=	Starting register to effect.
End&	=	Ending register to effect.

MS_Fadeout (R&, B&, G&, Speed&, Start&, End&)

Fades out to a single color.

R&,G&,B&	=	RGB Values to fade to (0-15).
Speed&	=	Speed (in VSYNCS) to fade out (0=fastest).
Start&	=	Starting register to effect.
End&	=	Ending register to effect.

MS_Fileselect (Path$, Flag&)

Calls up a GEM file selector and returns free disk space in MS_Var% and the selected file in MS_File$.

Path$	= Path to give to the file selector.
Flag&	= If this is 1 then disk space will be determined.

Note: This command will set the current drive path to the selected file if the file selected exists.

MS_Flashcolors

This procedure holds the default background flash colors.

MS_Flashinit

Installs the background flash colors and turns on the flash effect. Requires that you have built a Flash table in DATA statements. The flash command now cycles color 0 through this sequence of RGB values.

Data must be in hex format with one digit for each Red, Green and Blue color register values. Up to 50 values may be entered. The Data list **must** be terminated with &H1388.

Example:
```
RESTORE Color_Label !
Cycles from black up through
MS_FLASHINIT.
Color_Label: ! The Red colors.
DATA &H000
DATA &H100
DATA &HF00
DATA &H1388 ! End of list.
```

MS_Flashoff

Turns off the flash effect and restores the background color.

MS_Getpixel (X&, Y&, Screen%)

Returns the color of a selected pixel in MS_Var%. Faster and more flexible than the **point** command.

X&,Y&	= Pixel to test.
Screen%	= Address of the start of the screen that the pixel is stored in.

MS_GRAPHICS_ENGINE

The master call that executes all sprite drawing, animation, movement, tracking, flashing, color cycling etc... This call effects the MS_Logical% screen.

MS_Graphics_Engine_Init (Sprites&)

Starts the M.A.G.E. system. Must be called first in order for any other M.A.G.E. call to work correctly.

Sprites&	= Number of sprites you are going to use in this game. This number is Zero based (ie:5 = 6 sprites).

MS_Jstick1 (Sprite&, HSpeed&, VSpeed&, HMin&, HMax&, VMin&, Vmax&, Stick&)

Moves a sprite based on joystick direction. See the GFA Basic manual for joystick direction table.

Sprite&	= Sprite number to effect.
HSpeed&	= Horizontal pixels to move.
Vspeed&	= Vertical pixels to move.
HMin&,HMax&	= Horizontal boundaries.
VMin&,VMax&	= Vertical boundaries.
Stick&	= Stick Direction.

MS_Jstick2 (Sprite&, HSpeed&, VSpeed&, HMin&, HMax&, VMin&, VMax&, Stick&, Amount&)

Moves a multiple of sprites based on joystick direction.

Sprite&	= 1st sprite to effect.
HSpeed&	= Horizontal pixels to move.
Vspeed&	= Vertical pixels to move.
HMin&, HMax&	= Horizontal boundaries.
VMin&, VMax&	= Vertical boundaries.
Stick&	= Stick Direction.
Amount&	= Amount of sprites to move.

WARNING: The command will not move any sprites if the first sprite exceeds its boundaries.

MS_Jupdate (Sprite&,HSpeed&,VSpeed&,Amount&,Stick&)

Moves a group of sprites based on joystick direction. Unlike MS_Jstick2(), this command does not boundary check.

Sprite&	= 1st sprite to effect.
HSpeed&	= Horizontal pixels to move.
Vspeed&	= Vertical pixels to move.
Stick&	= Stick Direction.
Amount&	= Amount of sprites to move.

MS_Ldirection (Sprite&, Hspeed&, VSpeed&, Length|)

Moves a sprite in a direction for Length& steps but does not turn the sprite off when it is finished moving.

Sprite&	= Sprite to effect.
HSpeed&	= Horizontal pixels to travel each step.
VSpeed&	= Vertical pixels to travel each step.
Length|	= Steps to travel. 255 Does NOT invoke continuous movement.

MS_Loadcp1_1 (Path$)

Load in a .CP1 file into character bank #1.

Path$	= Path and filename of the .CP1 file.

MS_Loadcp1_2 (Path$)

Load in a .CP1 file into character bank #2. This bank MUST have the same dimensions as the first.

Path$	= Path and filename of the .CP1 file.

MS_Loadgpa (Path$)

Loads in an .GPA file.

Path$	= Path and Filename of the .GPA file.

MS_Loadmap (Path$)

Loads in a .MAP file.

Path$	= Path and Filename of .MAP file.

MS_Loadsp1_1 (path$)

Loads in a .SP1 file into sprite bank #1.

Path$	= Path and Filename of the .SP1 file.

MS_Loadsp1_2 (Path$)

Loads in a .SP1 file into sprite bank #2. This bank must have the same dimensions as the first bank.

Path$	= Path and Filename of the .SP1 file.

MS_Loffset (Sprite&, Offset&)

Sprite will follow an offset pattern. When offset is complete, the sprite will remain on. This command does **not** turn a sprite on.

Sprite&	= Sprite to effect.
Offset&	= Offset to use.

MS_Mapdown (Cells&, Screen%)

Displays the Character Map one cell lower. Updates the Character Screen and draws the appropriate images onto a physical screen. This command in fact moves the Character Screen "Window" one step lower on the Character Map.

Cells&	= Vertical cells to draw (MUST be at least 2).
Screen%	= Screen address to draw images onto.

WARNING: Going out of bounds of the .MAP file can cause a crash.

MS_Mapdraw (Cells&, Screen%)

Draws a section of the Character Map onto a screen and updates the Character Screen. The global Variables MS_Maph& and MS_Mapv& determine where in the Character Map you will draw from.

Cells&	= Vertical cells to draw (MUST be at least 2).
Screen%	= Screen address to draw images onto.

MS_Mapleft (Cells&, Screen%)

Displays the Character Map one cell to the left. Updates the Character Screen and draws the appropriate images onto a physical screen. This command in fact moves the Character Screen "Window" one step left on the Character Map.

Cells&	= Vertical cells to draw (MUST be at least 2).
Screen%	= Screen address to draw images onto.

WARNING: Going out of bounds of the .MAP file can cause a crash.

MS_Mapmove (SX&, SY&, DX&, DY&, Width&, Heigh&)

Moves a block of cells inside the Character Map. Does not update anything outside the Character Map.

SX&,SY&	= Upper left corner of source block.
DX&,DY&	= Upper left corner of destination block.
Width&	= Width of block.
Height&	= Height of block.

WARNING: This routine is **not** intelligent. You may not overlap the source/destination blocks.

MS_Mapright (Cells&, Screen%)

Displays the Character Map one cell to the right. Updates the Character Screen and draws the appropriate images onto a physical screen. This command in fact moves the Character Screen "Window" one step right on the Character Map.

Cells&	= Vertical cells to draw (MUST be at least 2).
Screen%	= Screen address to draw images onto.

WARNING: Going out of bounds of the .MAP file can cause a crash.

MS_Mapup (Cells&, Screen%)

Displays the Character Map one cell higher. Updates the Character Screen and draws the appropriate images onto a physical screen. This command in fact moves the Character Screen "Window" one step up on the Character Map.

Cells&	= Vertical cells to draw (**must** be at least 2).
Screen%	= Screen address to draw images onto.

MS_Mbitpeek (mX&, mY&)

Returns the value of a Character Map cell from bit map coordinates. This does not PEEK into the character screen but into the Character Map itself. Converts bit map location into nearest Character Map cell.

X&, Y&	= Bit map coordinates to PEEK.

MS_Movem (Source%, Destination%, Bytes&)

Block memory move. Much faster than GFA's BMOVE command.

Source%	= Source address of block.
Destination%	= Destination address of block.
Bytes&	= How many bytes to move. MUST be a number divisible by 4.

WARNING: This move is NOT intelligent. You MAY NOT overlap the source and destination block areas!

MS_Movesprites (Sprite&, HSpeed&, VSpeed&, Amount&)

Global sprite movement command.

Spirte&	= Starting sprite to move. This needs to be the **highest** number sprite because the routine works **downward**.
HSpeed&	= How many horizontal pixels to move.
VSpeed&	= How many vertical pixels to move.
Amount&	= How many sprites to move.

MS_Msk320 (Source%, Destination%, Lines&)

This routine creates a mask on the fly for two overlapping sections of screen memory, allowing for a 16 color overlay on top of a pre-existing screen. This routine is, of course, pretty slow. Both Source and Destination areas are assumed to have standard ST Low screen layouts (160 bytes per scan line). This routine copies two scan lines at a time, thus the 320 in the command name.

Source%	= Source screen address.
Destination%	= Destination screen address.
Lines&	= How many groups of 2 scan lines to copy and mask.

MS_Multi160 (Source%, Destination%, Width&,Lines&)

Copies a variable scan line-length screen onto a standard scan line length screen. Very powerful as it allows you to set up a page in memory that is two or more screens long (320, 480, etc.. bytes per scan line) and copy a section of this "Long" screen onto normal display screens. This allows for you to make the physical screen a "Window" into a much larger screen.

Source%	= Starting address of source screen.
Destination%	= Starting address of destination screen.
Width&	= Width of a source screen scan line.
Lines&	= How many scan lines to copy.

MS_Multi8msk (Source%, Destination%, SWidth&, DWidth&, Bytes&, Lines&)

Allows graphics blocks to be merged onto the screen. This command works much like MS_Multimsk320 but allows for only sections of a scan line to be copied and masked.

Source%	=	Start address of source block of memory.
Destination%	=	Start address of destination block of memory.
SWidth&	=	Width of source scan line in bytes.
DWidth&	=	Width of destination scan line in bytes.
Bytes&	=	Number of groups of 8 bytes to copy from each scan line.
Lines&	=	How many scan lines to copy.

MS_Multi8or (Source&, Dest&, SWidth&, DWidth&, Bytes&, Lines&)

Allows two memory blocks to be OR'd with each other. Great for multiple plane backgrounds. Care must be taken in selecting colors for proper operation.

Source%	=	Start address of source block of memory.
Destination%	=	Start address of destination block of memory.
SWidth&	=	Width of source scan line in bytes.
DWidth&	=	Width of destination scan line in bytes.
Bytes&	=	Number of groups of 8 bytes to copy from each scan line.
Lines&	=	How many scan lines to copy.

MS_Multimov (Source&, Destination&, SWidth&, DWidth&, Bytes&, Lines&)

Block move for odd numbers. Much slower than other MultiMov commands. Excellent when you need to copy sections of screens to each other. Does an overwrite, no Masking nor OR is preformed.

Source%	=	Start address of source block of memory.
Destination%	=	Start address of destination block of memory.
SWidth&	=	Width of source scan line in bytes.
DWidth&	=	Width of destination scan line in bytes.
Bytes&	=	Number of bytes to copy from each scan line.
Lines&	=	How many scan lines to copy.

MS_Multimov8 (Source%, Destination%, SWidth&, DWidth&, Bytes&, Lines&)

FAST copy of memory blocks between screens. Excellent for moving video "Windows" to the main display pages.

Source%	=	Start address of source block of memory.
Destination%	=	Start address of destination block of memory.
SWidth&	=	Width of source scan line in bytes.
DWidth&	=	Width of destination scan line in bytes.
Bytes&	=	Number of groups of 8 bytes to copy from each scan line.
Lines&	=	How many scan lines to copy.

MS_Offset (Sprite&, Offset&)

Makes a sprite follow an offset and turns the sprite off when the offset is finished. This command does NOT turn the sprite on.

Sprite&	= Sprite to affect.
Offset&	= Offset to activate.

MS_Offsetf (Sprite&, Offset&, Target&)

Makes a sprite follow an OffsetF based on a target sprite. The sprite will be turned off when the OffsetF is completed.

Sprite&	= Sprite to affect.
Offset&	= OffsetF to use.
Target&	= Sprite to preform OffsetF off of.

WARNING: Do **not** try to use an Offset in place of an OffsetF or unusual results **will** occur.

MS_Ord320 (Source%, Destination%, Lines&)

Quickly merges two 160 byte width scan line memory blocks (assumes screen memory) by OR'ing them together. Does two scan lines at a time.

Source%	= Start address of source memory area.
Destination%	= Start address of destination memory.
Lines&	= Number of groups of 2 scan lines to move.

MS_Pattern (Sprite&, Pattern&)

Makes a sprite move along a pattern. When the pattern is complete, the sprite is turned off.

Sprite&	= Sprite to affect.
Pattern&	= Pattern to follow.

MS_Peekmap (mX&, mY&)

Peeks a Character Map location. Returns the value of the cell in the global varible MS_VAR%.

mX&, mY& = Character Map coordinates.

MS_Plane1%, MS_Plane2%

C:MS_Plane1% (L: Source%, L: Destination%, W: Lines&)

Moves a single bit-plane of a pre-defined background screen to another screen. Typically this is used for quick parralax scrolling effects.

Source%	=	Source address for the start of the screen you want to copy from. This screen **must** be in standard Atari Low Resolution format!
Destination%	=	Destination address of the screen you want to write to. This screen must also be in Atari low resolution format.
Lines&	=	How many scan lines to copy.

MS_Pokemap (mX&, mY&, CHAR&)

Poke a Character Map cell location with a value.

mX&, mY&	=	Character Map cell to poke.
Char&	=	Value to poke into the character cell.

MS_Programinit (Flag&)

Starts up your program, including res-changes and various other functions such as turning off the mouse and activating the port 0 joystick.

Flag& = 0: The program will NOT use GEM (Default).
1: The program intends to use GEM. If this value is used, no resolution change may be preformed. GEM does not correctly change resolutions with the M.A.G.E. system. This mode is **not** recommended.

MS_Programend

Ends your program, restores desktop resolution and colors, re-activates the mouse and releases reserved memory banks.

MS_Resetfps

Resets an internal counter that tracks the number of Vsyncs that have passed since the last time this routine has been called.

MS_Resetsprites

Resets all sprites to their initial values, turns off all sprites and zeros their locations, animations ect...

MS_Resetvsync

Resets video memory locations to the same value. Used to de-activate page flipping.

MS_Restorepal

Restores desktop colors.

MS_Restoreworkpal

Restores a previously saved working pallette. Useful for restoring a pallette that will be modified during the main game loop.

MS_Roffset (Sprite&, Offset&)

Makes the sprite follow an Offset and loops the Offset instead of turning the sprite off when the end of the Offset is reached.

Sprite&	= Sprite to affect.
Offset&	= Offset to use.

MS_Roffsetf (Sprite&, OffsetF&, Target&)

Makes a sprite preform an offsetF which repeats when it is finished instead of turning off the sprite.

Sprite&	= Sprite to affect.
OffsetF&	= OffsetF to use.
Target&	= Target sprite to follow.

MS_Rpattern (Sprite&, Pattern&)

Makes a sprite follow a pattern and will repeat when the end of the pattern is reached instead of shutting the sprite off.

Sprite&	= Sprite to affect
Pattern&	= Pattern to use.

MS_Sampleplay (Adr%, Length&, Speed&)

Plays a sample recorded in ST-REPLAY format in the interrupt.

Adr%	= Address of the sample.
Length&	= Length of the sample in bytes.
Speed&	= Playback speed in HZ (3-15).

WARNING: This routine will CRASH on a Falcon. It is the only command in the M.A.G.E. which will not work on the FALCON. We recommend you either use DMA digital sound or use our own MODPLAY library.

MS_Samplestop

Stops a sample during re-play.

MS_Samplewait

Pauses the program until the sample finishes playing.

MS_Sbitpeek (Sprite&)

Peeks the Character Screen under the sprite's location and returns its value in the global varible MS_Var%.

Sprite&	=	Sprite to peek under.

MS_Sbitpoke (X&, Y&, Sprite&, Screen%)

Plots a sprite image on the screen and pokes its value into the nearest Character Screen cell.

X&, Y&	=	Bitmap coordinates to plot sprite image.
Sprite&	=	Sprite image to plot.
Screen%	=	Starting addresss of the screen where the image will be drawn on.

MS_Sbplot (MS1&, MS2&, MS3&, MS1%)

Plots a sprite at a given screen address using character cell coordinates. (For best results, Character and Sprites sizes should be the same.

MS1&	=	Horizontal Location of Character
MS2&	=	Vertical Location of Character
MS3&	=	Character
MS1%	=	Screen Address

Example: This routine can be used to save memory by allowing the sprites characters to plotted on the screen, thus emulating characters. (See Chapter 11 for an example)

MS_Sbtext (X&, Y&, String$, Screen%)

Plots characters on a screen using sprite images. Does not update the Character Screen.

X&, Y&	=	Position to start plotting.
String$	=	String to plot.
Screen%	=	Starting addresss of the screen where the images will be drawn on

MS_Screenadr (Bank&)

Returns the starting address of a screen bank in the global varible MS_Var%.

Bank&	=	Screen bank you want the address of.

MS_Screeninit (Banks&)

Reserves a number of full 32,032 byte screen and palette banks.

Banks&	=	Number of banks to reserve.

MS_Screento (Bank&, Flag&)

Copies the data in the MS_Physical% screen to a bank.

Bank&	=	Bank to load data into.
Flag&	=	0: Copy screen data only. 1: Copy Screen and Palette. 2: Copy Pallette only.

MS_Scroll96 (Source%, Destination%, Lines&)

Moves a 96 byte per scan line source screen to a 160 byte destination screen. Used to scroll over a smaller background map and display it on the main display. Many games use this technique.

Source%	=	Address of the first 96 byte wide scan line to be copied.
Destination%	=	Addresss of the start of the 160 byte per scanline screen.
Lines&	=	How many scan lines to copy.

MS_Scroll120 (Source%, Destination%, Lines&)

Moves a 120 byte per scan line source screen to a 160 byte destination screen. Used to scroll over a smaller background map and display it on the main display. Many games use this technique.

Source%	=	Address of the first 120 byte wide scan line to be copied.
Destination%	=	Addresss of the start of the 160 byte per scanline screen.
Lines&	=	How many scan lines to copy.

MS_Setvclip (Y&)

Sets the lowest Y point (the last scan line) that sprites will be display. Sprites will be clipped below this line.

Y& = Lowest point a sprite may be drawn.

MS_Showfps

Shows the estimated frames per second of the game that is currently running.

MS_Sound (Sound&, Block&, Rank&)

Plays one of the 101 built in M.A.G.E. sound effects.

Sound& =	Sound effect to play (0-100,101 is a blank sound).
Block& =	How many game loops before the next sound can play.
Rank& =	1: Play sound at once (it out RANKS any current sound).
	2: Wait until current sound is finshed playing.

MS_Splot (X&, Y&, Image&, Screen%)

Plots a sprite image on a screen. Does not update the Character Map or the Character Screen.

X&,Y& =	Location of image.
Image& =	Image to plot.
Screen% =	Start address of the screen image is to be plotted on.

MS_Sprite (Sprite&, X&, Y&, Image&)

Turns a sprite on and sets its location.

Sprite& =	Sprite to turn on.
X&,Y& =	Position of sprite.
Image& =	Sprite image to use to display sprite.

MS_Spritebank (Bank&)

Sets the active sprite bank. Bank must already be loaded or a crash may occur.

Bank&	=	Bank to set active (0 or 1).

MS_Spriteblock(Source%,X&,Y&,Width&, Lines&, Screen%)

Allows blitting of a graphic block to any X,Y coordinates with on the fly masking. In other words, it allows you to use large pre-drwan screen areas as super-size sprites. This routine can get a bit slow if extremely large blocks are used.

Source%	=	Starting address of source block of data.
X&,Y&	=	Screen cordianates to blit to.
Width&	=	Width of image in pixels.
Lines&	=	How many scan lines to copy.
Screen%	=	Starting address of the destination screen the image will be drawn on.

MS_Spriteclr (Sprite&)

Clears a single sprite and turns it off.

Sprite&	=	Sprite to clear.

MS_Spriteoff (Sprite&)

Turns a sprite off, does not clear its data.

Sprite&	=	Sprite to turn off.

MS_Spoke (cX&, cY&, Image&, Screen%)

Plots a sprite image on the screen and places the image number in the Character Screen cell.

cX&,cY&	=	Character Screen cell location.
Image&	=	Image to plot.
Screen%	=	Starting address of the screen the image is to be drawn to.

MS_Starinit (Stars&, HMin&, VMin&, HMax&, VMax&)

Sets up the Star plot arrays for the MS_Starplot command.

stars&	=	Stars to init (0 based).
HMin&	=	Left horizontal boundary to draw stars.
VMin&	=	Upper vertical boundary to draw stars.
HMax&	=	Right horizontal boundary to draw stars.
VMax&	=	Lower vertical boundary to draw stars.

NOTE: The boundaries do not de-activate stars (like the sprite boundaries do) instead they simply place the star at the beginning of the opposite boundary.

MS_Starplot (Stick&, Screen%)

Moves and plots the Star arrays based on joystick direction.

Stick&	=	Stick Direction. See GFA Basic manual for details.
Screen%	=	Starting address of the screen stars will be plotted on.

MS_Starset (Star&, X&, Y&, Color&, HDir&, VDir&)

Sets a single star's parameters.

Star&	=	Star number to modify.
X&, Y&	=	Star position.
Color&	=	Color of the star.
HDir&, VDir&	=	Magnitude of movement MS_Starplot() should use. This value may not be less than zero.

MS_Stepmode

Pauses the program until a keystroke is detected. Useful in de-bugging only.

MS_Storepal

Stores the current palette in the PAL buffer.

MS_Storeworkpal

Stores the current palette in the **Workpal** buffer.

MS_Track (Sprite&, HSpeed&, VSpeed&, Length|, Target&)

Makes a sprite track another sprite. Turns sprite off after length expires.

Sprite&	=	Sprite to affect.
HSpeed&	=	Horizontal pixels to move per loop.
VSpeed&	=	Vertical pixels to move per loop.
Length|	=	How many game loops to track (255 = continuous tracking).
Target&	=	Sprite to track.

MS_Vsync (Catch&)

Page flips only. Does not re-draw the MS_Logical screen.

Catch&	=	How often to skip a Vsync (0= Flicker free page flipping).

MS_Vsync8 (Lines&, Catch&)

Page flips and draws 8 scan line groups from the MS_Background% screen to the MS_Logical% screen.

Lines&	=	How many groups of 8 scan lines to copy.
Catch&	=	How often to skip a Vsync.

MS_VsynC10 (Lines&, Catch&)

Page flips and copies 10 scan line groups from the MS_BACKGROUND% screen to the MS_LOGICAL% screen.

Lines&	=	How many groups of 10 scan lines to copy.
Catch&	=	How often to skip a Vsync.

MS_VsynC20 (Lines&, Catch&)

Page flips and copies 20 scan line groups from the MS_Background% screen to the MS_Logical% screen.

Lines&	= How many groups of 20 scan lines to copy.
Catch&	= How often to skip a Vsync.

MS_Vsyncdelay (Vsyncs&)

Pause the program until a certain number of Vsyncs have passed.

Vsyncs& = How many Vsyncs should pass.

MS_Vsyncdraw (Source%, Bytes&, Length&, Catch&)

Page flips and redraws from a user defined background screen.

Source%	=	Source address of data to copy from.
Bytes&	=	How many bytes down MS_LOGICAL% to start drawing (16,000=half).
Length&	=	How many bytes to copy.
Catch&	=	How often to skip a Vsync.

MS_Waitkey

Wait until a key is pressed and return its value in the global varible MS_Var%.

MS_Waitstick (Joystick&)

Wait until a joystick button is pressed.

Joystick& = Joystick button to monitor (0 or 1).

MS_Waitvsync (Speed&)

This command is used in conjunction with MS_Resetfps to cause the program to run at a set number of Vsyncs. This command only sets the **top** speed of your program.

Speed& = Number of Vsyncs per game loop. (Frame re-draw).

MOD Player Interface Expansion Commands

MS_Init_Player (Mods&)

Initializes the Mod Player and sets up the command buffers for re-playing your previously saved MOD files. **Must** be called before any other MOD commands are used.

Mods& = Number of Mods you will be using in your program.

MS_Play_Mod (Command$, Mod&, Attribute&)

Controls all the aspects of Mod re-play and control.

Command$ = Holds a letter that designates the function to perform on the mod. A list of commands follows:

"S"	= Start playing a Mod.
"O"	= Turn a mod Off.
"+"	= Speed up your mod.
"-"	= Slow down your mod.
"<"	= Turn volume down.
">"	= Turn volume up.

Mod&	=	Mod buffer to effect. If buffer is empty then the program may crash.
Attribute&	=	When playing a mod pass a 1 if you want the player to repeat the song or a 2 if you want this mod to play only once.

MS_Fade_In (Speed&)

Not to be confused with MS_FadeIn(). This routine fades in a currently playing mod to full volume at a selected speed.

Speed& = Speed to fade in (0=fastest).

MS_Fade_Out (Speed&)

Not to be confused with MS_FadeOut(). This routine fades out a currently playing mod to no volume at a selected speed.

Speed& = Speed to fade out (0=Fastest).

Falcon030 Interface Command Set

MS_Rezchange

Forces **all** Atari machines to Low resolution Atari ST mode. Exception to this is a ST/STE in High res (which cannot be forced into low res Atari Mode without slowing the machine down beyond all usefullness. If the exception is encountered, then an alert will be displayed and program execution terminated.

MS_Changeres (VAR Falcon&, Oldrez%)

Support routine for the MS_Rezchange routine. It does some of the actual leg work involved in switching the resolution around. Not normally called by the user.

Falcon&	= Has a Falcon been detected?
Oldrez%	= Old resolution/operating code - see developer docs.

MS_Returnrez (Falcon&, Oldrez%)

Returns the machine to the original resolution and mode it was operating in before the program took control.
BEWARE: Never pass any values to this routine that MS_Changeres() did not set.

Falcon&	= Restore via Falcon commands.
Oldrez%	= Old resolution to return to.

MS_Cookie (Cookie$, VAR Cookie&, Cookie1&)

A powerful tool for looking at the cookie jar. The routine is self documented and is used by the MS_Changeres() routine. Users should not have to access this routine.

Cookie$	= Segment of cookie jar to search for.
Cookie&	= Upper word of cookie data (-1 if no jar present).
Cookie1&	= Lower word of cookie data.

Chapter 11. The 20 Minute Game

This section is geared toward one thing only, designing a very simplistic game for tutorial purposes. Hopefully this will be your first look at a M.A.G.E. game. This section does not **explain** the M.A.G.E. Shell routines and how they work. This is done elsewhere. This section just explains how they can be used in conjuction with the M.A.G.E. Shell.

Our quick and easy 20 Minute Game will have the following features:

Game speed of 17 frames a second in 50hz, 20 frames in 60hz video.
25 16 X 10 sprites
Sprite animations
Pattern handling
Direction movement
Explosion routines
Main player designed using 2 sprites
Main player controlled by the MS_Jstick2( ) procedure
Enemy missile shooting
Character graphics text display, star field

NOTE: A clever technique is used in this game to save data storage. The sprites, text, and the star displays were all designed in 1 sprite image file. For the text and numbers and other characters we had to first designate a black color other than the background. Next we fill in the holes (empty pixels) in these special characters so that when they are plotted (as sprites) they still erase what they are plotted over. This emulates the plotting of regular characters! Nice...and sneaky!!

Total of 3 different types of levels. (Pattern, direction, random)

This game was created by Dave Munsie, the original creator of the M.A.G.E.. The following comments are by him:

When putting the finished M.A.G.E. product together I realized we did not have any simplistic, tutorial type games as part of the main package. While many of the included games are of a very high quality, they did not really show what could be done in a few short minutes of time. I challenged myself to write a game that is playable and somewhat fun to play that could be programmed in under a half an hour. Using the M.A.G.E. Shell really allowed me to finish this game in under 30 minutes. The finished game shows very accurate ways to handle main play collisions that occur **out of the main loop** along with proper ways to init and end a game. There are also some very easy methods to handling the score display. My job with this section is not to explain the M.A.G.E. Shell procedures but only the code that I had to create and why. I will explain the whole program from top to bottom, but I am only going into detail on the code that I had to write for this 20 Minute Game.

The first step to designing this game was thinking about what it was going to be. This took about 10 seconds. I decided on creating a **simple-shoot-em-up** game that uses MS_Patterns to control all the nasties. All I really needed to do now was make the bad guys shoot at the main sprite and have the main sprite shoot back. Throw in a couple of collision checks, text handling and I am done! So I put together a few custom sprites and stole a few ideas from Rob. I then created a few animation sequences and a total of 5 patterns with the MS_Anim editor. This took about a total of 5 minutes. I created a generic charcter set with a few custom cells to represent the ground and the main sprite so I could display how many ships are left in the game. Another 5 minutes are gone. So now I have about 10 minutes to throw the game together!

One thing nice about the M.A.G.E. Shell is that it really makes first time game programmers feel at ease by already designing the **procedure** names and letting you just fill in the blanks. I only had to add one custom routine to finish the game. Everything else was just added **into** the specific M.A.G.E. Shell procedures.

So! About 10 minutes left eh? Okay, let us take a look at the sections of the game where I had to add something to from **top** to **bottom** as they would appear in the actual source code.

MS_graphics_engine_init (24, 0)

This is at the top of the shell to initialize how many sprites I wanted to use. I figured 25 (0-24) sprites should allow me to write a game that would run at 3 Vsyncs from the interpreter as well as when compiled, (Do not forget things fly when compiled). The second number, 0, tells the shell I do not want the extra 30k of memory used for the MS_Blits. They are fast but I do not need them for this game.

```
PROCEDURE title_screen.
MS_fadeout (0, 0, 0, 0, 0, 15) ! Fade all to black.
MS_resetvsync ! Always reset this at this procedure.
MS_finished&=FALSE ! Needed for title loop.
'
' Okay, a few title screen messages:
'
MS_Sbtext (0, 0," 30 MINUTE DEMO GAME", MS_physical%)
MS_Sbtext (0, 1X10, "     BY DAVE MUNSIE", MS_physical%)
MS_Sbtext (0, 4X10," PRESS FIRE TO PLAY!", MS_physical%)
MS_Sbtext (0, 17X10, "  PRESS ESC TO EXIT", MS_physical%)
'
MS_fadein (0, 0, 0, 0, 15) ! Okay, fade in the text.
'
REPEAT
'
MS_finished&=STRIG(1) ! Poll the joystick button.
'
IF BIOS (&H1, &H2) ! Any keyboard data waiting?
keyboard_control ! Let the design shell handle it.
```

```
ENDIF
'
UNTIL MS_finished&
'
MS_clearstick(1) ! Clear the joystick button.
'
RETURN
```

Most of this code was already in the shell. I just had to add the text and fades! Okay, next section...Init Game.

```
Procedure init_game. ! All code is set up once per game start.
'
MS_lives&=3 ! How many lives to give player per game.
MS_score%=0 ! Reset global ms_score% counter.
'
MS_resetsprites ! Clear all sprites and re-init system pointers.
MS_shmin&=-1 ! Screen boundaries for sprites.
MS_shmax&=320 ! Screen boundaries for upper left corner.
MS_svmin&=-9
MS_svmax&=180
MS_setvclip(180) ! Set clip at bottom scan line for full screen animation.
'
'
level&=0 ! What level are we on?
eanim&=0 ! Enemy animation type.
epat&=0 ! Enemy pattern type.
level_type&=1 ! Enemy style of movement.
'
init_level ! Any level variables that need to be initialized.
init_sprites ! Maybe this could be used to set up the initial sprites.
init_gamescreen ! Gosub here to actually draw the first background screen.
'
Return
```

As you can, see not much user code has actually been generated so far. Next, we come to the init_game screen section.

```
PROCEDURE init_gamescreen ! Initializes first game screen.
'
' This ensures a clean graphic startup without any screen flicker.
'
MS_fadeout (0, 0, 0, 0, 0, 15) ! Fade to black.
'
MS_resetvsync ! Always at start up.
'
' Step 1: Draw any background data to ms_background%.
'
' Hmmm, my first problem...backdrop? Simple character stars.
'
FOR x&=1 to 50
MS_sbplot (RANDOM(20)+1, RANDOM(18), RANDOM(4)+18, MSbackground%) ! Stars
NEXT x&
'
'MS_Sbtext(0, 18X10, String$ (20, 23),MS_Background%) ! Land.
'
' Step 2: Copy everything from MS_background to MS_logical.
'
MS_copy_background_to_logical (0, 0, 200)
'
' Step 3: Draw any graphics that are not redrawn every loop.
'
MS_Sbtext (0, 19 X 10, "0", MS_logical%) ! Starting score.
MS_Sbtext (17 X 16, 19 X 10, STRING$ (3, 22), MS_logical%) ! Draw 3 ships to start with.
MS_DrawSprites (MS_logical%)
'
' Step 4: Finally copy everything from MS_Logical% to MS_Physical%
'
MS_copy_logical_to_physical (0, 0, 200)
'
```

```
MS_fadein (0, 0, 0, 0, 15)
'
RETURN
```

There is nothing really too technical here. The sta background was quick and dirty, but I only have about 1 minutes left!! Now we come to the procedure that is called a the beginning of every game level.

```
PROCEDURE init_level
'
Enemy_total_hit&=0 !  How many killed per level.
'
FOR x&=17 DOWNTO 0 !  Clear enemy-shot flags.
enemy_shot&(x&)=0
NEXT x&
'
'INC eanim& !  We only have four enemy animations.
IF eanim&>4
eanim&=1
ENDIF
'
INC epat& !  We only have 5 different patterns available.
IF epat&>5
epat&=1
ENDIF !  Every level use a different enemy animation.
'
INC level& !  Once we go past 2 levels...time for RANDOM.
IF level&>2
level_type&=RANDOM(3)+1
ENDIF
'
RETURN
```

This procedure just resets the enemy shot counters and changes the patterns and animations per level. One interesting thing is that when all 5 patterns have been sucessfully destroyed, the program kicks in the random mode.

Level_type& can be from 1-3.
1 = Enemies follow pattern.
2 = Enemies are directioned controlled.
3 = Enemies have different anims and patterns.

This allows a little variety and the illusion of multiple attack waves. Sneaky, and actually not that bad for a desperate move!

Coming up next, the init_sprites routine, gets called at the start of every game and after the main player gets hit by a nasty!!

```
PROCEDURE init_sprites
'
MS_resetsprites
MS_sprite(0,150,170,8) ! Init the main shooter.
MS_sprite(1,166,170,9) ! Created from 2 sprites.
enemy&=2
'
RETURN
'
```

Clears all sprites. Then sets up a 32 X 10 object made from two 16 X 10 sprites. Enemy&=2 just tells me which nasty gets moved first!

Travelling down the source code trail brings us to the game over routine. Short, sweet and to the point. Okay, I will do a little scroll up effect.

```
PROCEDURE game_over ! Called when the game is over.
'
MS_clear (MS_background%)
MS_Sbtext (6 X 16, 0, "GAME OVER", MS_background%)
temp%=MS_physical%+(88 X 160)
MS_sound (61, 0, 1) ! A little bounce sound.
FOR x&=1 to 10
Vsync
MS_movem (MS_background%, temp%, x& X 160)
SUB temp%, 160
```

```
NEXT x&
MS_vsyncdelay(180)
'
RETURN
```

The trick here is the MS_movem( ) command. Notice the source area is stationary at pointing to the top of the background screen where I drew the game over message. The key thing here is the number of bytes moved from this location (x&*160). With each loop (1-10), more information will be moved from this location. And since I actually move the destination address (temp%) up a scan line per loop, the end result is that the words "GAME OVER!" will seem to grow upwards from the middile of the screen!! Finally I let the user say SHOOT, GAME OVER for about 3 seconds.

Next up with about seven minutes left is probably the hardest part about this game. Actually launching and controlling the sprites. We will break this down into three problem areas:

1) Move main player back and forth and allow him/her to shoot.
2) Launch enemy nasties using one of three styles. (level&=1-3)
3) Make enemies shoot a missile every once in a while.

```
Procedure update_players ! Called from the main game loop.
'
'Let's tackle these one at a time...
'
'1) Move main player back and forth and allow him/her to
shoot.
'
'Remeber our main sprite is actually designed using two sprites.
'We have to update both of these every loop. The MS_jstick2( )
'command will take the whole movement problem in one fast
'command.
'
```

```
MS_jstick2 (0, 5, 0, -1, 305, 160, 160, STICK(1),2) ! Auto control
of main sprite ship.
'
'Our main player is made from sprites 0 - 1.
'
'How do we make the player fire a missile?
'
IF STRIG(1) ! Joystick button pressed?
'
IF MS_sflag|(2) = 0 ! Launch a missile at enemy.
MS_Sprite|(2)=10
MS_Shor&(2)=MS_Shor&(0)+7
MS_sver& (2)=MS_Sver&(0)-4
MS_direction (2, 0, -8, 25) ! Move verically up 8 pixels at a time.
MS_sound (30, 0, 0) ! A simple sound.
ENDIF
'
ENDIF
'
'Here we have to control the nasties based on what type of level
'it is: Patterns, direction movement, random patterns etc...
'
IF level_type&=1
INC spacing&
IF spacinig&=6
spacing&=0
INC enemy&
IF enemy&>17
enemy&=3
ENDIF
IF MS_sflag|(enemy&)=0 AND enemy_shot&(enemy&)=0
MS_anim (enemy&, eanim&)
MS_rpattern (enemy&, epat&)
ENDIF
ENDIF
'
ELSE
'
```

```
IF level_type&=2 ! Direction move, vertically.
INC enemy&
IF enemy&>17
enemy&=3
ENDIF
IF MS_sflag|(enemy&)=0 AND enemy_shot&(enemy&)=0
MS_animate (enemy&, eanim&, RANDOM (320), -8)
MS_direction (enemy&, 0, RANDOM(5)+1, 200)
ENDIF
ELSE
'
INC spacing&
IF spacing&=6
spacing&=0
INC enemy&
IF enemy&>17
enemy&=3
ENDIF
IF MS_sflag|(enemy&)=0 AND enemy_shot&(enemy&)=0
MS_anim (enemy&, RANDOM(3)+2)
MS_pattern (enemy&, RANDOM(4)+1)
ENDIF
ENDIF
ENDIF
ENDIF
'
'Make enemies shoot a missile every once in a while.
'The idea is to make the missiles shoot out of one of the nasties.
'This can be done rather easily using random statements.
'
IF RANDOM(200)<100 ! Fire missiles at main player.
missile&=18+RANDOM(7)
IF MS_sflag|(missile&)=0
miss_enemy&=3+RANDOM(15)
IF MS_sflag|(miss_enemy&)=1
MS_anims|(missile&)=0
MS_shor& (missile&)=MS_shor& (miss_enemy&)
MS_sver&(missile&)=MS_sver&(nuss_enemy&)
```

```
MS_Sprite|(missile&)=11
MS_direction (missile&, 0, RANDOM(6)+2, 180)
ENDIF
ENDIF
ENDIF
'
RETURN
```

The trickiest parts were handling the three different levels of nasty movements. I usually recommened something like: ON level_type& GOSUB level 1, level 2, level 3, etc...etc... But this game had to be done quickly, so I had to think **on the fly** rather than plan ahead of time.

Look close between level 1 and 3 and you will see that level 1 launches a sprite using a **Rpattern.** This will cause the sprites to repeat this pattern until it is shot. In level 3 we just use **Pattern**, this enables us to re-use the sprite using a **Random** pattern each time it finishes a pattern. We also **randomize** the animation style.

Level 2 shows how easy it is to make several sprites fall down the screen. By passing **Random(5)+1** vertical pixels to move we can cause the sprites to fall at different speeds.

In the section that fires a missile you will see how easy it is to launch a sprite from another sprites location. By passing another sprite's location directly to the procedure, we can easily accomplish this task.

The next section is dealing with what happens when the missile hits an enemy sprite and what happens when any enemy sprite hits our main player, (which is actually two sprites wide).

```
Procedure collision_check ! Called from the main game loop.
'
MS_collide (2, 0, 0, 15, 9, 0, 0, 15, 9, 17, 15)
'
```

```
IF MS_var% ! Any collision?
'
MS_sound (3, 20, 1)
MS_explode (MS_var%, 5, 6) ! Make this sprite explode.
MS_spriteoff(2)
'
ADD MS_score%, 50
INC enemy_total_hit&
enemy_shot& (MS_var%)=1
IF enemy_total_hit&=15
init_level
ENDIF
'
MS_Sbtext (0, 19 X 10, STR$ (MS_score%), MS_logical%)
MS_copy_logical_to_physical (190, 190, 10)
'
ENDIF
'
MS_collide (0, 2, 0, 28, 9, 0, 0, 15, 8, 24, 22)
IF MS_var% ! Any value>0 means an enemy hit them.
'
player_hit ! We do not care which enemy it was.
'
ENDIF
'
RETURN
```

Note the use of the array enemy_shot&( ). This holds the status to tell if an enemy has been shot during a level. If this value=0, then we should launch the sprite otherwise, we assume it has already been shot. In the collision routine we set it to 1 when it has been hit by the missile. Notice we do not allow the enemy missiles to be shot down, which saves us a collision detection.

Otherwise nothing really too hard to comprehend here. The MS_collide( ) Routine deserves your attention though. This is probably the most difficult thing to grasp when designing games with the M.A.G.E.

Well, only a few minutes left! Luckily we have just one more routine to write and that is the main player collision routine.

```
PROCEDURE player_hit
MS_graphics_engine ! Used to finish the current frame.
DEC MS_lives&
MS_Sbtext (17 X 16, 19 X 10, STR$ (MS_lives&,22)+STR$ (3-MS_lives&, 32), MS_logical%)
MS_copy_logical_to_physical (190, 190, 10)
MS_direction (0, -7, 0, 50)
MS_direction (1, 7, 0, 50)
FOR x&=18 to 24
MS_animate (x&, 5, MS_shor&(0)+RANDOM(16), MS_sver&(0)+RANDOM(3))
MS_direction (x&, -3+RANDOM(7), -RANDOM(3), 80)
NEXT x&
'
MS_sound(8,0,1)
'
FOR x&=1 to 90
MS_resetfps ! Reset Vblank timer.
MS_vsync20 (9, 0) ! Draw 180 scan lines from background.
MS_graphics_engine ! Let the engine do its thing.
MS_waitvsync(2) ! Wait until Vblanks passed 2.
NEXT x&
'
IF MS_lives& ! Any lives left? If so, re-init level.
'
MS_fadeout (0, 0, 0, 0, 0, 15)
'
init_sprites
'
FOR x&=17 downto 0
enemy_shot&(x&)=0
NEXT x&
```

```
enemy_total_hit&=0
'
MS_movem (MS_background%, MS_logical%, 28800)
MS_movem (MS_background%, MS_physical%, 28800)
MS_drawsprites (MS_logical%)
MS_drawsprites (MS_physical%)
'
MS_fadein (0, 0, 0, 0, 15)
MS_vsyncdelay (20) ! Wait a little while to collect your thoughts.
'
ENDIF
'
RETURN
```

This last routine is important because it gives a nice **clean** and structured way to handle animations outside of the main game loop. The method as shown above will allow you to create bonus and special anims that happen during the main loop. It also has possibilities for end of game routines. Again, nothing tricky shown here. The method of using a **fixed for next loop** allows you to precisely determine the length of the routine. Also, fading the background to black before resetting all the sprites allows a more professional look to the game.

Well, that is it! This game did indeed only take 20 minutes to write. But a lot of you might be wondering a few things if you turned to this section right away, so a few explanations are in order:

20 minutes....HOW?

The M.A.G.E. Shell has all the routines already created and structured in a way that makes it very easy to just **fill** in the blanks with the code that I needed for this game. The only routine that I **had to add** was the **Player_Hit** routine. Everything else was already there for me to fill in with my own code. Where is the main loop? Remember, the only sections I was supposed to write about was the ones I had to program. The main loop was left untouched because I did not have to add anything to it (Yes, the M.A.G.E. Shell does a lot in letting me design this game in 20 minutes)!

Chapter 12.

The Creation of "Thurg N Murg"

I have been asked to describe how I have created "Thurg N Murg", giving you a step by step description of how the game creation process on the M.A.G.E. system worked to produce such results. I have declined in this chapter to describe the actual mechanics of the M.A.G.E. system. Instead, I will go over major points that should save you many hours of hair pulling frustrations.

The Game Idea

The first step to creating a game is to get a clear idea of what that game is going to do. My first step was to decide what type of game I was going to design. I then sat down at the kitchen table and wrote down an outline of what I wanted "Thurg N Murg" to be. Here is a condensed version of what I wrote:

Game concept - This will be a single screen at a time platform game in which the main character can jump, run and fire. The enemy can do the same things as the player and will have different levels of intelligence. I want the game to have a nice soundtrack, a full set of musical sound effects and smooth joystick response. The game will run at a target rate of 30 FPS and will be allowed to slow down as far as playability will allow. The player will be able to pick up extra powers and lots of goodies. A timer will swallow entrance to the bonus goodie screeen.

Pickups - A bunch of typical pickups that I will assign values to.
Powers - Extra Jump Power, Extra Fire Power, Extra Speed and Shields.

Characters - My character set will hold two fonts and some backgorunds. Characters will be 16 x 10 and the sprites will be 16 x 20. I chose these sizes because the characters can make more flexible backgrounds and the large sprites allow me to avoid using multiple segmented sprites.

Sprites - Since I am wasting some time plotting large sprites for shots, I will keep collision checks to a minumim. There will be two players with three shots each which makes for no more than eight collision checks per frame.

Background checks will use standard CPEEK and CPOKE commands and will be programmed with check-specific options in mind.

The lower twenty scan lines will be used for scores, timer, and lives display. These displays will be laid out ahead of time in Deluxe Paint ST.

Sound a little chaotic? Well, the this process always is. After I composed a brainstorming list, I fired up my ST and drew a mock-up screen. This screen was drawn displaying the maximum amount of graphics that I envisioned the game using at one time. I drew all displays with lives maxed out, scores at all 9's and the timer and bonus powers at full display. This is not an easy process folks! It requires hours of tinkering and fiddling. But once you have a completed screen that you can feel comfortable with, then you have a great programming tool in that screen. By the way, I have used pre-drawn sprites in this game. If you want to design your own sprites, then there are two steps you can take at this point. You can draw all the graphics you expect to use and then draw your mock up screen or you can just use filled blocks of the same size that your sprites are going to be. I highly recommend drawing out your sprites first because it gives you a clear idea of what your final product will look like.

Once I had a mock-up screen designed, I noted all important screen coordinates on a piece of paper. This included the start location of each display (ie: lives, score, ect..). I also noted the space between each figure in the display. I also noted what I would need to draw each part of the display (sprites or characters) and any conflicts between displays. I also noted where my game screen started and ended, and what possible boundary problems I might have.

Now that I had designed my mock-up screen, I put together my sprite and character banks. I took note of all important sprite numbers (like the sprites I used for extra live displays). I noted most important character types; it was at this point that I decided to use a free floating background designation. I decided that the upper left corner character of each screen will be a legal background piece, all other characters on the screen would be illegal moves (platform pieces). Once I had decided this, I made my MAP files. I experimented with the backgrounds as some of them did not turn out as I expected. Note the difference between my .PC1 files and what actually made it to the character bank. Such experimenting is essential to getting the correct look to your game. Also, do not get depressed if you can not get that look in one session. It took me two days to get my character set the way I wanted it.

Now I put together my animations. I realized that a frame of animation I had captured for falling did not work as well as it was planned to so, I deleted them and used the slots for other things. This is why my sprite banks have a slightly chaotic look to them. If you realize that I loaded in a nice set of sprites (as per the .PC1 file) and then played with the animations and then fiddled with the sprite bank, you will understand how the bank got that way.

I then pasted together my animations. Actually this happened in a sort of twisted semi-simultaneous time frame as the sprite bank construction. The animations are almost never what you want the first time around, so do not get depressed if you need to come back here quite often.

The last thing to do before programming was the sound effects. I am lucky in the fact that I have a musician who makes music for me. My soundtracks were already waiting for me. If you do not have this luxury, then feel free to curse my name as you design your own soundtracks. My suggestion is that you bribe a musician buddy at school for the required licks. I then thought of all the possible actions all the characters would make and planned sound effects around them. It is important to note that some sound effects are better left **out**. For instance, it would be easy to give every monster its own jumping sound but since the monsters are almost **always** jumping the user would hear almost nothing but that single sound effect throughout the game. Making the sound effects was a matter of loading in a sound and tweaking it until it became what I needed. I then created a MOD file with exactly one note on three channels and used the MOD interface to play back the sound effects later.

Notice how I have not programmed a single thing yet? It is not an accident! Structured programming requires that you get all of your ground work done first. The more elaborate the game, the longer the above process will take. "Thurg N Murg" took two days and I am probably the most experienced M.A.G.E. user around. Do not get frustrated if it takes you several days to complete these steps, as it will pay off in an **awesome** product!

Oops! I almost forgot another **important** aspect of getting ready. I took notes on everything and I mean **everything!** Before I sat down to actually write "Thurg N Murg" I had all of my support files ready and 6 typed pages of notes (with lots of handwritten sidebars). Typical things I kept handy were my .GPA printouts, all the sprites that I used to display the bonus pickups, pickup values in the 2nd character map, pickup bonus numbers, sprite frame numbers, what main varibles would control important stuff (like Thurg&() and Monster&() - I actually laid out how these arrays were going to look before I wrote the program). Anything I would need to know about the data I had just created was in a little folder that I guarded with my life. Now, you will not know what all to keep track of on your first game, so do not expect to have all your bases covered all the time. Do **not** worry that you have got to backtrack to the editors once in awhile to get an important sprite frame or a character number you forgot. Just make sure you have got some serious **notes**. The difference between a good programmer and a great one is the preperation involved.

I will harp on a few more things right now that, if you listen to me, will save you a major headache. **Always** use descriptive variable names for important variables. Only use single letter variables when you are using that variable to perform a trash calculation. **Always** keep notes. Learn to use those note taking and planning skills as you develop your programming. If you do, your games will begin to come together like clockwork. Last but not least, **always finish your game.** The biggest problem with a lot of programmers today is that they will not finish their game. They tinker and tinker or start a new program before the current project is complete. Do not fall into this trap! No matter how bad the project looks, at least get your current code running before re-writing it. Never pursue more projects than you can comfortably handle. I am really guilty of breaking that rule and I have paid for it!

Now I can actually program the game. Since the code is pretty straight foward and you have other, more nuts-n-bolts examples to pour through, I will not get into much detail here. Instead, I will discuss the points of interest throughout the game. What follows is a topic-headed discussion of each major point of interest in the code:

Title Sequence: The reason I have short names here, is at first, I planned to have a series of routines handle a different run-through. However, T2 worked out so well that I decided to keep it and build other data routines to handle the title sequence. The high score tables were added at the last minute (or my beta-testers would have lynched me!), so that is why it looks like a patch (it is one!). The title sequence also shows how you can use an on gosub structure and some counters to control a bunch of different things. I personally hate to do title screen work, so mine tend to be bland. Yours could really be great!

Player Logic: Okay, since I never know what the player is going to do from moment to moment I use a status flag to determine which routine should handle the movement. Different routines include: move left (S4), move right (s8), start jump (s1), handle Jump and Fall (s2). Look at what each of these procedures do. They all follow this basic pattern: Is the move legal? If not, then adjust status flag and position to a legal position and change animation. If it is legal, continue movement as desired and do not change animation. The player should stay within bounds and should not be able to go through a platform.

Check_Legal: I use these routines in many of my programs. Sometimes I chain a whole table to this routine (ie: I give each of the 200 cells a legal (0) and illegal (1) value). In this case, I have only one type of cell per screen that is legal. I use this to return a legal value. The different types of Check Legal are just hard-wired to check specific areas depending on what I want to look at. Hard wiring small routines like this is a very good way to get better program speed.

Init_Playscreen: This routine looks very complex at first but if you break it down it makes much more sense. The first portion simply determines the start of the screen in the Character Map. If you count on your fingers and follow the logic (by putting in different values of current level&) you see that this code keeps init_level& from 0 to 29 and the X&, Y& positions correct.

Next the routine draws the map onto the background screen and sets the single legal character background piece by peeking 0, 0. I set this up this way in order to make any piece the background and I have to do nothing except that the cell 0, 0 in the Character Map is the background value. We now set up the background some more by drawing the goodies, timer and clearing the players' power icons.

Now the code sets up the monsters. Feel free to weight this any way you want. Basically, each little If-Then block checks to see if the level is high enough to introduce this monster type to the game and then attempts to set these monsters up. As a suggested improvement (I am doing these in my next game!), I would say make the INIT_monster routines be a single routine that reads each monster's type of attributes from a table.

Now the program gets a small patch that enforces child mode. I never put child mode in until the program is pretty much finished. That way I know exactly where I can modify some varibles to achieve the effect I want.

Place Goodies: I wanted to talk about this routine because it could be kind of confusing to an inexperienced programmer. Since I am working with a 16 x 10 character set and a 16 x 20 character set, I can not plot directly above a platform. Now the value of this goodie (assigned by myself and has no importance outside the collision check routine) is plotted into only **one** of the "Fake" Character Screen cells. If this confuses you, do not worry. Play with the Sub Y&,2 value to see what I am doing. The rest of each little block (each block lays down a different type of goodie) justifies the character cell into screen coordinates so that I can plot it. Experienced eyes may ask why I can not do this through the M.A.G.E. commands? The answer is simple, the sprite and character heights are different. But the MS_Splot routines take the character values as the true values so I have to do a manual justification in order to correct for this.

Place Nasties: A lot of this routine is a patch. I added some hidden screens for some depth. The thing to point out here is that I do a lot of variable manupliation that could be cleaned way up if I had taken more time and used tables. Sorry, but they gave me a week to write this whole program.

The Fall Tables: These tables control all movement during the game (except for plain vanilla horizontal). The fall tables have a variable length but they all end in 5 (Fast fall). Every character on the screen keeps a pointer to the fall table it is currently using. This pointer is increased unless the value it is pointing to is equal to 5. You will see me use some pretty complex looking array referencing with the fall tables but they all follow the same logic:

1) Add the vertical value I am currently point to to my sprite position.
2) Am I pointing at the number 5?
3) No - Increase Pointer value.
   Yes - Do not do anything else!

You can use this same logic to construct some pretty wild movement. The reason to use hand-coded tables and not offsets is that the hand coded tables can sometimes be more accurate and flexible than a plain old offset. Try altering the fireball paths (you can do curley ques or sine waves or anything else!! I am planing to use whirlwind shots in my sequal - "Son of Thurg N Murg").

Overview: I want to give you a quick overview of what I do during a typical game loop.

- Move the players based on Max_players and joystick input.
- Move shots.
- Move all active monsters.
- Move those neat little bonus numbers.
- Collide everything.
- Do it again until game is over or level is complete!!

As a last note, I would like to discuss a few things you **should** play with. Mess with these values and routines and watch how this effects the game.

- Play with the speeds. I have clearly marked where speeds are affected. Try making the game run slower or faster. Violate the 16 pixel boundary and see why you should not.
- Play with the Fall paths. This makes for some great variations on the original game.
- Make some monsters more intelligent or less intelligent. How about a **really** smart Mondo monkey?
- Try plotting less or more goodies. Be warned..the program can go into an infinite loop if it can not find an open slot.
- Change the maps around. Design new and weird maps. See why I designed the maps the way I did (some designs do not work well!).
- The monster intelligence routines are woefully underpowered. Try writing in some checks to make them smarter. Hint: Control jumping a little better or keep the character from dropping off just above the player, which makes them harder to kill.

If anybody has any questions about some of the code just mail me the questions and I will be happy to discuss anything not covered in this chapter! Put "Attention: Robert Dytmire" on the envelope below MajicSoft's address.

Thanks,

Rob

MajicSoft®

"We Put The Majic In The Software!"

Chapter 13. The Creation of "Sleuth"

A. Layout and Design of "Sleuth"

Looking back over the years of computers and computer games, there are always a couple of games that really stand out amongst the crowd. These games are the ones that are now classics in their own right.

When I first sat down with the M.A.G.E. and tried to decide what type of game I wanted to design - my mind raced back through the years, looking for an idea that would spark creativity in me. Out of all of the arcade games I have ever played, a game from a long dead company named Synapse, came to mind. It was a game that a friend and I spent many months playing. A game that formed an everlasting friendship between my friend Steve and I because of all of the hours spent playing and mapping it. That game was "Shamus" and it was written by a man named William Mataga. At the time we were playing the game, I owned an Atari 400 with 16K of memory and the games were supplied on a cartridge. Steve and I got so addicted to "Shamus" that we used to rush home during lunch hour at work to play it. And after work, we would rush home to play it some more. Both Steve and I were married at the time, and I think "Shamus" became a much hated game to our wives, because the wives took second place to it. Steve and I practically ignored them during the three months it took to map and finish the game. We even snapped off a Polaroid picture of the last screen. That screen being the one in which we destroyed the Shadow.

In today's arcade games much has been lost to fancy graphics and sound effects. Today's programmers seem to put more into the looks and sounds of games than into the play ability of them. The addictiveness, simplicity and shear enjoyment have been lost. With that thought in mind, I decided to do a Shamus clone for my first arcade game. The first and most important part of my design was to design a game that is simple and fun to play. The graphics and sound were second. To do this, "Shamus" became my role model.

My Atari 400 and all of the software I had collected through the years had be given away many years ago. My friend Steve however, kept his old Atari and software, so I contacted him and asked if MajicSoft could borrow his computer and "Shamus" cartridge. I explained that we were going to design an Atari ST version of the game. He eagerly agreed to help and sent his machine and game to me via "Next Day Air" by UPS. "Shamus" was also his most favorite game from the old Atari 8-Bit days.

The package came the next day and I ripped open the box. Boy was I excited. It had been so long since I had seen one of these old Atari computers. Playing that old "Shamus" game really brought back some great memories. They just do not make them like that anymore.

After playing the game for a few hours to get the feel of it again, my creative juices really started flowing. So, I shut the machine off and headed to my office to boot up my Atari ST computer. The first step in designing a game was now behind me, that being to decide what type of game I wanted to design. The next step was to load up my paint program and start roughing out the graphics. Due to the way the M.A.G.E. Character editor uses certain sizes in characters, I chose a grid layout of 16 X 10 pixels. This small grid allows for greater detail. This grid is what I used to draw all of my character pieces in. The game was to have four levels in it, so I drew four different colored wall pieces. I also wanted to have a hint system built into the game so I drew four different fill patterns. I duplicated these patterns four times and colored them with the four colors I had chosen for the levels.

I continued drawing the game pieces until I had everything I needed and then saved off the pictures in a PC1 format. The next step was to load the M.A.G.E. Character editor and grab the different pieces from the pictures I had drawn to be used in my character set. This character set is the one I used to create the map. The Character Editor has the ability to save off two different types of files. One being the CP1 file and the other being the SP1 file. The CP1 file is the character file. It is used in the Map Editor to draw the maps with. The SP1 file is used in the Sprite Editor to make sprites with. I first grabbed all of the pieces that would be used in the Map Editor and saved the file as SLEUTH.CP1. I used SLEUTH because that is the name I had chosen for the name for the game.

The next step was to design the game map. Since this was the first time I had ever used the M.A.G.E., I really wanted to push it to the limit just to find out what its limitations were. So with graph paper and pencil I drew a 16 X 16 map grid and then drew the map within the grid. Next I loaded the M.A.G.E. Map Editor and loaded in the SLEUTH.CP1 and started to design my game map. This step took the better part of two days. A 16 X 16 map yields two hundred and fifty six rooms! With the map finally complete, I saved off the file as SLEUTH.MAP. This was the final step for me in the designing of "Sleuth" game. All of the files and pictures were next turned over to Larry Scholz for his part, the actual programming of the game. Larry's tutorial follows. I hope that you enjoy the final result.

Sincerely, John Stewart.

B. The Programming of "Sleuth"

When John Stewart called me up and asked me to write a game with the M.A.G.E. I was very excited. I had seen several of the shareware games created with this graphics engine and I wondered what I could do with it. At this point I had never used the M.A.G.E. before. I was wondering if I could learn the M.A.G.E. and write the program in the time he gave me, which was about two weeks. John suggested that we do a clone of

Shamus, a classic Atari 8-bit game. Since this was one of my favorite games years ago I was doubly excited. John and I discussed what graphics were needed from him before I could start programming.

For the next couple of days while I was waiting for the graphics I started to familiarize myself with the 3 main utilities supplied with the M.A.G.E.. These are the Character/Sprite Editor, Map Editor and Animation Editor. I loaded each utility up and ran it through its paces. I made sure that I understood the operation of each. After a couple of days I got the call from John saying that the graphics and map are ready. I quickly called up the company Bulletin Board Service (BBS) and downloaded the graphics files being anxious to get started.

The first thing I did was to look at the pictures that I would turn into sprites. Since there were animation sequences for every sprite, you can imagine how it must of looked to me at first. I then loaded up the Character Editor so I could start grabbing sprites. The first thing I noticed was that John had drawn everything with a background behind it. I think this is a very common mistake to watch out for. The sprite image must **not** have anything around it. By right clicking on the image from the editor I removed all of the background. So now I have the following images that I will use as sprites:

Animation Sequences: Sleuth walking (left, right, up, down), Hemroids, Spiroids, Hopperoids, Spirit, explosion, plus a special Sleuth dieing and Spirit dieing for end of game.

Aditional Images: 8 of Sleuth's bullets (one for each direction), 8 of monster's bullets, Spirit bullets, 2 sets of numbers from 0 to 9. one in green and the other red, a set minature keys to display at bottom of screen, the minature Sleuth image to indicate remaining lives.

One important thing to remember if you are going to use numbers or letters in your sprite set: They must appear in their proper positions, which is as they appear in the Atari character set. This means that the number 0 must start in position 48 followed by 1 - 9. Since I needed two different colors of numbers I needed to do something a little different. The next available area was 49 spaces ahead of the first number set. When I use them in the M.A.G.E., I now have to add 49 to each position in the number string so that the routine for printing text using sprites will find the new numbers. The reason I used the sprite set instead of using the character set is because of one important factor. When you use the character set the number will replace everything that was previously on the screen under that number. When using sprites they will blend in with the background and not erase it. Since I had a 10 pixel high character set and a 8 pixel high box to put the numbers in, I really had no choice.

Now that I had all of my sprite images, the next step was to load up the Animation Editor to create the actual animations. I made an animation from each of the sequences mentioned above. I also made one pattern for the Spirit to follow at the end of the game. I saved my animations off as a SLEUTH.GPA file.

Now I have the following files created:

• SLEUTH.MAP	created by John
• SLEUTH.CP1	created by John
• SLEUTH.SP1	created by me
• SLEUTH.GPA	created by me

These 4 files were updated many times during the creation of Sleuth. You will probably need to make several changes to get everything just right. Next comes the programming of the game. The first thing I had to decide on was how to set up my sprites. I finally decided on the following:

0	Sleuth
1-2	Sleuth's bullets
5-8	Monster's bullets
9-13	Hemroids
14	Hemroid that fires bullets
16-20	Spiroids
21	Spiroid that fires bullets
23-27	Hopperoids
30	Spirit

You will notice that I left room for improvements such as more bullets for Sleuth, an extra Hemroid that shoots, an extra Spiroid that shoots and a couple extra Hopperoids that do something.

Now I am ready to load up the M.A.G.E.. I will load in the SP1, CP1, MAP, GPA files from disk for now until I am sure I am done changing them. When I am finished changing them, I will use the Inline Maker utility and turn them into **inline** data.

The first programming I did was to have the first room of the map fade in on the screen. From there I got Sleuth walking around on that screen. In this game the walls are deadly and will kill Sleuth. So the next thing I did was to check to see if Sleuth walked into a wall. I did this by checking the map data ahead of the sprite with the MS_cpeek routine. This routine will return the value of the character you point to. If this character is not a floor piece or a special pick-up, such as a key or a potion, then he dies. So now that I have Sleuth walking around on the screen, the next step was to make him walk from room to room. Every time Sleuth takes a step I have to check his position to see if he is leaving a room.

By the time I got this far it was apparent that it is easy to **lock up** the screen. This will happen many times until you are familiar with the M.A.G.E.. The screen will go black and stay that way or everything just stops moving. However, the good old 3 key salute usually works just great. Hitting **<control>** **<alt>** **<left shift>** and then **<return>** will return you to the Editor 99.9% of the time. You should pay close attention to the type of error that occurred and the line above the cursor which contains the error.

Now that I have Sleuth walking around from room to room, I walk him through all the rooms just to make sure they were designed correctly. I found a few tight spots that needed fixing so I fixed them in the Map Editor.

Now for a little action. The next thing I did was to make Sleuth shoot. Sleuth can only have two bullets on the screen at one time. Every time I push the trigger I check the variable MS_sflag|(sprite no) to see if one of the two bullet sprites I have assigned to Sleuth are available. If MS_sflag|() returns a '0' then that sprite is available. If the timer says it is time to fire another bullet, then I do so. He can only fire his bullets at a certain speed. For firing the bullets I used the MS_direction routine. This routine will send the bullet sprite off in any direction you set up. Every game loop I check to see if a bullet sprite has been activated by using the MS_sflag|() routine again. If it is active I check to see what map character is under the bullet. If it is a wall piece, I shut the bullet sprite off. This is done so the bullets do not go through the walls.

The next step I decided on was to let Sleuth pick up objects like keys, potions and question marks. With each step that Sleuth takes, I am already checking what is under his feed, so the next step will be easy. I look up in my notes the character values of all of the objects. If one of those objects is under Sleuth's feet, I then draw floor pieces over the object. This makes the object disappear. I then take the appropriate actions depending on which object Sleuth picks up. I keep track of what colored keys are picked up. If Sleuth walks over a keyhole and he has the same colored key, I open the gate. I determine where in the room Sleuth is so that I know which gate to open. I replace gate pieces with floor pieces on the map to make the gate open.

Now that Sleuth can defend himself, we needed some bad guys for target practice. There are 3 types of enemies, plus the Spirit. The enemies needed to appear randomly around each room. As Sleuth enters each room the number of enemies is determined by how much exploring Sleuth has done up to that time. The enemies have certain boundries they can appear in so they are not right on you when you enter a room. I also did not want them appearing in walls. Again, since my sprites are 16 X 20 and the characters in my map are 16 X 10 I needed to do two checks every time I placed an enemy.

To make the enemies move was the next step. Every game loop I would check each enemy sprite to see if it is active. If it is active then I pick a random direction to move it. I first check to see if it is clear in that direction. If there are no walls then I update that monsters position. The monsters do not walk through walls either.

Time for target practice now. To check to see if one of Sleuth's bullets hit an enemy I use the MS_collide() routine. I use this routine once each game loop. If a bullet sprite is active I check to see if it collided with an enemy. The MS_collide() routine will return in MS_var% the sprite number collided with or a 0 if no sprite was hit. If a sprite is hit, I turn off the bullet sprite and call the MS_explode() routine. This routine will replace the enemy sprite with an explosion animation which I had set up earlier. As you can tell from the discription, the enemies are not very smart. The dumbest enemies are the Hemroids. They will only start heading towards you if you get close enough. The Spiroids have a little more intelligence and start tracking you from a greater distance. Finally, comes the Hopperoids. These monsters are special. They stand in one spot and do a quick hop and then disappear. They then reappear two spaces closer to you. They lock on to you the second you walk into a room. These were the trickiest monsters to move. What I did with these monsters was to check first to see if they were active. If they were active, I would then check to see which frame the animation was on. I would do that with the MS_anifcnt() variable. This tells which frame the animation is on. When it was on the final frame then I would move it. Since these monsters move two spaces at a time and I did not want them walking through walls, I had to check both spaces ahead of them. If there is a wall then they just move up to the wall.

Now that all the monsters are moving around nicely it was time to add special monsters that would shoot back at Sleuth. It was already pretty hectic in a room if there were a lot of monsters, so I did not want to add too many shooters. I chose to add possibly one Hemroid and one Spiroid. There is a 50% chance that a shooter will appear. I performed the same collision check with the enemy bullets as I did with Sleuth's bullets.

Now for more collision checking. Every game loop I also need to check to see if Sleuth ran into any of the enemies. Now I have 4 collision checks each game loop. They are:

1. Check to see if Sleuth hits an enemy.
2. Check to see if one of Sleuth's bullets hit an enemy.
3. Check to see if any of Sleuth's bullets hit an enemy bullet.
4. Check to see if an enemy bullet hits Sleuth.

The reason I have to have so many collision checks is that I am checking against different size sprites in each collision check. In the first one I am checking a 16 X 20 sprite against other 16 X 20 sprites. In the second one I am checking a 3 X 3 sprite against other 16 X 20 sprites. In the third check I am checking a 3 X 3 sprite against other 3 X 3 sprites. Finally, in the last one I am checking 3 X 3 sprite against a 16 X 20 sprite. The second and last collision check look the same, but they are not. There are different sprites that are being checked against.

The last enemy I had to deal with was the Spirit. The Spirit only comes into a room if you are in there for a certain amount of time. A **gong** will sound giving you a three second warning. When the Spirit comes he will start in the upper corner farthest away from you at the time. He will lock onto you and track right to you. He will go through walls or anything else he has to in order to get to you. The only defence against him is, if you hit him with a bullet, he will freeze for a couple of seconds before he resumes his attack. If he touches you, you lose a life.

Now that everything is moving and shooting I need to update a few displays. I need to show the remaining lives in the upper right corner display. This display will hold up to 15 little Sleuth sprites. I need to show the high score above the player's score in the upper left corner. I also need to show the room number, elapsed time and posessions along the bottom of the screen. The score and elapsed time need to be updated every game loop. I plotted these on the screen using sprites. I talked earlier about how I did the numbers, which I used for the scores and elapsed time.

About the only thing missing now is some nice sound effects. I used the MS_sound() routine for all of my sound effects. To do this I did ran the program and then gave it the old 3 finger salute to stop it. This would not let the program end normally, so that I could go into direct mode by hitting the ESC key. From here I would type in MS_sound(1, 0, 1) and **[Return]**. This plays the sound for me. I then went through all 100 sounds, making a list of which ones I liked for the various effects.

Now that everything was the way I liked it, I loaded up the Inline Maker program. Up to this time the SLEUTH.CP1, SLEUTH.SP1, SLEUTH.MAP, SLEUTH.GPA files were being loaded in off of disk. I created **inline** data out of each of them. I then had these four files: SLEUTH.CGP, SLEUTH.SGP, SLEUTH.MGP, SLEUTH.GGP. Next, I brought these files into the program with the inline command. This was the last step. The program was finished and ready for inclusion into the M.A.G.E. box set.

This project took me a little less than two weeks to complete from start to finish and I had never used the M.A.G.E. before. This should encourage you! Do not be discouraged when you see the number of procedures the M.A.G.E. uses. Jump right in and start your own project. Feel free to experiment with the source code to Sleuth. Try changing variables and see what happens. The only way to learn is to experiment. I hope you enjoy playing this game as much as I did writing it. Thank you!

Larry Scholz

Technical Support

M.A.G.E. comes with free technical support. In order to receive technical support, you must send in your warranty card. If your warranty card is not on file at MajicSoft, you will not receive any support, so please take a moment right now to fill out your card and mail it to us.

When contacting MajicSoft for technical support, the following will save time and help us give you a quick answer:

- Please insure you have read the relevant sections of the manual carefully and the READ.ME file (if one exists) before contacting us (many problems are simple mistakes which could easily be avoided by referring to the manual).

- Give us both the version number of the software from the About M.A.G.E. dialog box and your exact hardware configuration. We will also need your disk 1 serial number. You may find it useful to make a note of it here:

Serial No.

- When telephoning, call while you are actually sitting in front of your computer if possible so that you can quickly try out suggestions we may make. Alternately, have the manual close by for reference.

Our technical support line is available Monday - Friday, from 9:00 am til 5:00 pm PST in the United States. The telephone number is (818) 701-1473. Our sales line is (803) 788-8177 and is open from 9:00 am til 5:00 pm EST in the United States.

Notes

Notes

MajicSoft®

"We Put The Majic In The Software!"