– APPENDIX C –

NATIVE FILE FORMATS

The .GEM File Format

Files ending in '.GEM' are graphic metafiles created by **GDOS**. They are usually used to represent vector graphics but may also be used to store links to bitmap images and textual information.

Two primary versions of **GEM** files exist. Version 1 files are guaranteed not to contain bezier curves whereas version 3 files may. Version 3.xx files are also commonly referred to as **GEM/3** files.

The Metafile Header

GEM metafiles begin with a header as follows:

WORD	Contents
0	Magic number (0xFFFF).
1	Header length in **WORD**s.
2	Version number (major * 100 + minor).
3	NDC Flag as follows:
	Value **Meaning**
	0 (0, 0) in lower-left corner (NDC)
	2 (0, 0) in upper-left corner (RC)
4	Minimum X extent.
5	Minimum Y extent.
6	Maximum X extent.
7	Maximum Y extent.
8	Page width in tenths of millimeters.
9	Page height in tenths of millimeters.
10	Lower Left X value of coordinate system.
11	Lower Left Y value of coordinate system.
12	Upper Right X value of coordinate system.
13	Upper Right Y value of coordinate system.
...	Other information may appear in the header following which is currently undefined. Use **WORD** #1 to skip any unknown information.

The definition of **WORD**s 4–13 is defined by the creator of the file using three metafile commands. **WORD**s 4–7 are set with the **v_meta_extents()** function. **WORD**s 8–9 are defined with the **vm_pagesize()** function. **WORD**s 10–13 are defined with **vm_coords()**. If the creator fails to specify defaults for any of these values, the appropriate values will be set to 0 in the header. If zeros appear for **WORD**s 10–13, the default NDC coordinate system should be assumed.

Metafile Records

Following the header will appear a list of records of varying length which, when translated, can be 'played back' on the destination **VDI** device. Each record is formatted as follows:

WORD	Meaning
0	Opcode of **VDI** function.
1	Number of *PTSIN* elements.
2	Number of *INTIN* elements.
3	Function sub-ID.
4...	*PTSIN* elements.
...	*INTIN* elements.

The list of records is terminated with an opcode of 0xFFFF (this record is written when a **v_clswk()** call is made by the creator).

When playing back **GEM** files, the application must translate all coordinates from the metafile coordinate system to that of the destination device. In addition, text metrics should be appropriately converted. If an unknown opcode is discovered it should be played after any elements of the *PTSIN* array are translated (making the assumption that they should be).

Metafile Sub-Opcodes

GEM metafiles support the use of special sub-opcodes for implementing reserved and user-defined functions. **GEM** metafile translators should ignore sub-opcodes they don't understand. Each sub-opcode can be identified with the primary opcode of 5, function ID of 99 and the first (required) member of *INTIN* being the sub-opcode ID. The currently defined sub-opcodes are as follows:

INTIN[0]	Meaning
10	Start Group.
11	End Group.
49	Set No Line Style.
50	Set Attribute Shadow On.
51	Set Attribute Shadow Off.
80	Start Draw Area Type Primitive.
81	End Draw Area Type Primitive.

None of the pre-defined sub-opcodes use additional *INTIN* or *PTSIN* elements though user-defined sub-opcodes may.

Opcodes from 0–100 are reserved for use by Atari. Sub-opcodes from 101-65535 are available for use by developers but should be registered with Atari to avoid possible conflicts.

The .IMG File Format

The IMG file format was designed to support raster images with a varying number of planes. In practice, almost all IMG files currently available are simple black and white single plane images because the original file format did not specify a method of storing palette information with the file. To fill this need, several unofficial extensions to the format were put into use (some of which were incorrectly implemented by applications supporting them). The color extension which will be discussed here to cover color images is the ‘XIMG’ format.

The IMG Header

Image headers consist of at least 8 **WORD**s as follows:

WORD	Meaning
0	Image file version (Usually 0x0001).
1	Header length in **WORD**s.
2	Number of planes.
3	Pattern definition length.
4	Source device pixel width (in microns).
5	Source device pixel height (in microns).
6	Scan line width (in pixels).
7	Number of scan lines.

Some IMG files will have additional header information which should be skipped or interpreted as discussed below.

Interpreting Extra Palette Information

If **WORD** #2 is set to 1, then the image data consists of one plane (i.e. monochrome) and any extra header information should be ignored.

If **WORD** #2 is set to 16 or 24 then the image data consists of that many planes of high color or true color data and any extra header information should be ignored. In a high color image, planes appear in the order RRRRR GGGGGG BBBBB. In a true-color image, planes appear in the order RRRRRRRR GGGGGGGG BBBBBBBB.

If **WORD** #2 is set to 2, 4, or 8, the image consists of palette based color image data. If no extra header information is given then the creator did not specify palette data for this image. If extra header **WORD**s appears they may be useful in determining the color palette. The two primary extensions to the IMG format are ‘XIMG’ and ‘STTT’. ‘STTT’ will not be discussed here as it does not serve well as a machine or device independent format. The ‘XIMG’ header extension is as follows:

WORD	Meaning
8 & 9	ASCII ‘XIMG’
10	Color format (Almost always 0 – RGB).
11...	RGB **WORD** triplets. Three **WORD**s appear for each pen. There are (2 ^ *numplanes*) pens. Each word contains a value from 0 to 1000 for direct passage to **vs_color()**.

Image Data Format

Each scanline contains data in **VDI** device independent format which must be converted using the **VDI** call **vr_trnfm()**. Each scanline is padded to the nearest byte. Every plane for each scanline should appear prior to the beginning of data for the next scanline. This allows interpreters to decompress and transform the image data a scanline at a time to conserve on time and memory. A sample ordering for a four-plane image is listed below:

Scanline #0 – Plane #0
Scanline #0 – Plane #1
Scanline #0 – Plane #2
Scanline #0 – Plane #3
Scanline #1 – Plane #0
Scanline #1 – Plane #1
Scanline #1 – Plane #2
Scanline #1 – Plane #3
etc.

Image Compression

Each scanline is individually compressed. This means that compression codes should not transgress over scanline boundaries. This enables decompression routines to work scanline by scanline.

Scanline data should consist of two components, a vertical replication count and encoded scanline data. In practice, however, some older .IMG files may not contain a vertical replication count for each scan line.

The vertical replication count specifies the number of times the following scanline data should be used to replicate an image row. It is formatted as follows:

BYTE	Contents
0	0x00
1	0x00
2	0xFF
3	Replication Count

Immediately following the vertical replication count is the encoded scanline data. This run-length encoding can by looking for three separate flag **BYTE**s. A 0x80 **BYTE** indicates the beginning of a bit-string item. A bit-string item is formatted as follows:

BYTE	Contents
0	0x80
1	Byte count ‘n’.
2...	‘n’ **BYTE**s of unencoded data.

A pattern-run item begins with a **BYTE** of 0x00. It specifies a fixed number of times that the pattern which follows it should be repeated. It is formatted as follows:

BYTE	Contents
0	0x00
1	Length of run.
2...	Pattern bytes (length of pattern is determined by header **WORD** #3).

Finally, a solid-run item begins with any other **BYTE** code. If the high order bit is set then this indicates a run of black pixels, otherwise it indicates a run of white pixels. The lower 7 bits of the byte indicates the length of the run in bytes. For example a **BYTE** code of 0x83 indicates a run of 24 black pixels (3 bytes).

The .FNT File Format

Filenames ending with the extension ‘.FNT’ represent bitmap font files. These files may be utilized by loading them through any version of **GDOS**. FNT files are composed of a file header, font data, a character offset table, and (optionally) a horizontal offset table.

The FNT Header

Font files begin with a header 88 **BYTE**s long. **WORD** and **LONG** format entries in the header must be byte-swapped as they appear in Intel (‘Little Endian’) format. The font header is formatted as follows:

BYTE(s)	Contents	Related VDI Call
0 – 1	Face ID (must be unique).	**vqt_name()**
2 – 3	Face size (in points).	**vst_point()**
4 – 35	Face name.	**vqt_name()**
36 – 37	Lowest character index in face (usually 32 for disk-loaded fonts).	**vqt_fontinfo()**
38 – 39	Highest character index in face.	**vqt_fontinfo()**
40 – 41	Top line distance expressed as a positive offset from baseline.	**vqt_fontinfo()**
42 – 43	Ascent line distance expressed as a positive offset from baseline.	**vqt_fontinfo()**
44 – 45	Half line distance expressed as a positive offset from baseline.	**vqt_fontinfo()**
46 – 47	Descent line distance expressed as a positive offset from baseline.	**vqt_fontinfo()**

48 – 49	Bottom line distance expressed as a positive offset from baseline.	**vqt_fontinfo()**
50 – 51	Width of the widest character.	N/A
52 – 53	Width of the widest character cell.	**vqt_fontinfo()**
54 – 55	Left offset.	**vqt_fontinfo()**
56 – 57	Right offset.	**vqt_fontinfo()**
58 – 59	Thickening size (in pixels).	**vqt_fontinfo()**
60 – 61	Underline size (in pixels).	**vqt_fontinfo()**
62 – 63	Lightening mask (used to eliminate pixels, usually 0x5555).	N/A
64 – 65	Skewing mask (rotated to determine when to perform additional rotation on a character when skewing, usually 0x5555).	N/A
66 – 67	Font flags as follows: **<u>Bit</u>** **<u>Meaning (if Set)</u>** 0 Contains System Font 1 Horizontal Offset Tables should be used. 2 Font data need not be byte-swapped. 3 Font is mono-spaced.	N/A
68 – 71	Offset from start of file to horizontal offset table.	**vqt_width()**
72 – 75	Offset from start of file to character offset table.	**vqt_width()**
76 – 79	Offset from start of file to font data.	N/A
80 – 81	Form width (in bytes).	N/A
82 – 83	Form height (in scanlines).	N/A
84 – 87	Pointer to the next font (set by **GDOS** after loading).	N/A

Font Data

The binary font data is arranged on a single raster form. The raster's height is the same as the font's height. The raster's width is the sum of the character width's padded to end on a **WORD** boundary.

There is no padding between characters. Each character may overlap **BYTE** boundaries. Only the last character in a font is padded to make the width of the form end on an even **WORD** boundary.

If bit #2 of the font flags header item is cleared, each **WORD** in the font data must be byte-swapped.

Character Offset Table

The Character Offset Table is an array of **WORD**s which specifies the distance (in pixels) from the previous character to the next. The first entry is the distance from the start of the raster form to the left side of the first character. One succeeding entry follows for each character in the font yielding (number of characters + 1) entries in the table. Each entry must be byte-swapped as it appears in Intel ('Little Endian') format.

Horizontal Offset Table

The Horizontal Offset Table is an optional array of positive or negative **WORD** values which when added to the values in the character offset table yield the true spacing information for each character. One entry appears in the table for each character. This table is not often used.

The .RSC File Format

Resource files contain application specific data which is generally loaded at run-time. RSC files contain **OBJECT** trees (see the discussion of the **OBJECT** structure in *Chapter 6: **AES***), strings, and images.

Two resource file formats are currently supported. **TOS** versions less than 4.0 support the original RSC format while **TOS** 4.0 and greater will now support the older format and a new extensible format. The original format will be discussed first followed by an explanation of the changes incurred by the newer format.

The RSC Header

Resource files begin with an 18 **WORD** header as follows:

WORD	Field Name	Contents
0	*rsh_vrsn*	Contains the version number of the resource file. This value is 0x0000 or 0x0001 in old format RSC files and has the third bit set (i.e. 0x0004) in the new file format.
1	*rsh_object*	Contains an offset from the beginning of the file to the **OBJECT** structures.
2	*rsh_tedinfo*	Contains an offset from the beginning of the file to the **TEDINFO** structures.
3	*rsh_iconblk*	Contains an offset from the beginning of the file to the **ICONBLK** structures.
4	*rsh_bitblk*	Contains an offset from the beginning of the file to the **BITBLK** structures.
5	*rsh_frstr*	Contains an offset from the beginning of the file to the string pointer table.
6	*rsh_string*	Contains an offset from the beginning of the file to the string data.
7	*rsh_imdata*	Contains an offset from the beginning of the file to the image data.
8	*rsh_frimg*	Contains an offset from the beginning of the file to the image pointer table.

9	*rsh_trindex*	Contains an offset from the beginning of the file to the tree pointer table.
10	*rsh_nobs*	Number of **OBJECT**s in the file.
11	*rsh_ntree*	Number of object trees in the file.
12	*rsh_nted*	Number of **TEDINFO**s in the file.
13	*rsh_nib*	Number of **ICONBLK**s in the file.
14	*rsh_nbb*	Number of **BITBLK**s in the file.
15	*rsh_nstring*	Number of free strings in the file.
16	*rsh_nimages*	Number of free images in the file.
17	*rsh_rssize*	Size of the resource file (in bytes). Note that this is the size of the old format resource file. If the newer format file is being used then this value can be used as an offset to the extension array.

Many of the header entries represent offsets from the beginning of the file. These offsets are expressed as positive unsigned **WORD**s making the standard file a maximum size of 64k bytes.

Object Trees

Each RSC file may contain a number of object trees. *rsh_object* contains an offset from the beginning of the file to the object trees (stored consecutively). The **LONG** array pointed to by *rsh_trindex* can be used to separate the object trees in the list. There are *rsh_ntree* **LONG**s in this array. Each array entry can be used as an array index to a different object tree. After being loaded in memory by **rsrc_load()**, the members at *rsh_trindex* are filled in with the absolute pointers to their respective trees.

Each individual **OBJECT** is stored differently on disk then in memory. In the file, pointers to **TEDINFO**s, **BITBLK**s, and **ICONBLK**s are stored as absolute indexes into the arrays of these members stored in the file. Therefore a **G_TEXT OBJECT** whose *ob_spec* field would normally point a **TEDINFO** in memory would contain the value 0 if that **TEDINFO** were the first **TEDINFO** contained in the file.

String pointers are represented on disk by their absolute offset from the beginning of the file. Image pointers in **BITBLK** and **ICONBLK** structures are likewise pointed to through absolute file offsets, not indexes.

Free Strings and Images

rsh_frstr points to a table of **LONG**s which each specify an offset from the start of the RSC file to a free string. *rsh_frimg* points to a table of **LONG**s which each specify an offset from the start of the file to a **BITBLK** structure.

AES 3.30 Resource Format

Beginning with **AES** 3.30, the resource file format was altered to allow for new **OBJECT** types. The only **OBJECT** which currently takes advantage of this format is **G_CICON**. **G_CICON**s can only be stored in files of the new format. The new format can be identified by the third bit of *rsh_vrsn* being set.

The Extension Array

Immediately following the old resource data (using *rsh_rssize* as an offset) an extension array is added. The first entry in this array is a **LONG** containing the true size of the RSC file. Notice that values such as these are now stored as **LONG**s to allow the size of RSC files to exceed 64k. Due to the method in which some older resource elements were stored many components of RSC files will still be constrained to 64k.

Following the file size is a **LONG** word for each extension present followed by a 0L which terminates the array. Currently only one extension exists (**CICONBLK**) and it *always* occupies the first extension slot. As additional extensions are added, a value of -1L for any entry will indicate that there are no resource elements of that type in the file. For example an extension array that does contain **CICONBLK**s would look like this.

...basic resource file...
LONG *filesize*
LONG *cicon_offset*
0L

The CICONBLK Extension

The **G_CICON** object type adds the ability to display color icons from the **AES**. The *ob_spec* of the object indexes a **CICONBLK** structure stored in the extension area. Each **CICONBLK** must contain a monochrome icon and a color icon for as many different resolutions as desired. When drawn, the **AES** will pick the icon that is the closest match for the current screen display. If there is no color icon present which the **AES** is able to convert, the monochrome icon is displayed.

The *cicon_offset* pointer gives an offset from the beginning of the resource file to a file segment which contains the **CICON** data. This segment contains a **CICONBLK** pointer table followed by the actual **CICONBLK**s.

The **CICONBLK** pointer table is simply a longword 0L for each **CICONBLK** present in the file. These pointers are filled in by the **AES** when loaded. The list is terminated by a -1L.

Immediately following the pointer table is one of the following variable length structures for each **CICONBLK**:

```
ICONBLK monoicon;     /* This is the standard monochrome resource. */
LONG n_cicons;        /* Number of CICONs of different resolutions. */
WORD mono_data[x];    /* Monochrome bitmap data. */
WORD mono_mask[x];    /* Monochrome bitmap mask. */
CHAR icon_text[12];   /* Icon text (maximum of 12 characters). */

/* for each resolution supported (n_cicons) include the following structure */

WORD num_planes;      /* Number of planes this icon was intended for */
LONG col_data;        /* Placeholder (calculated upon loading). */
LONG col_mask;        /* Placeholder (calculated upon loading). */
LONG sel_data;        /* Placeholder (must be non-zero if ‘selected’ data exists */
LONG sel_mask;        /* Placeholder (calculated upon loadind). */
LONG next_res;        /* 1L = more icons follow */
WORD color_data[n];   /* n WORDs of image data (n is num_planes*WORDs in mono
       icon).*/
WORD color_mask[n];   /* n WORDs of image mask. */
WORD select_data[n];  /* Only present if sel_data is non-zero. */
WORD select_mask[n];  /* Only present if sel_data is non-zero. */
```

CICON Images

All color image data is stored in **VDI** device independent format on disk and is automatically converted by **vr_trnfm()** upon **rsrc_load()**[1].

[1]Due to a bug in some versions of the **VDI** the seventh **WORD** of color icon image data may not contain the value 0x0001. If it does, the **VDI** may incorrectly display the icon.