– CHAPTER 7 –

VDI

Overview

The Virtual Device Interface (**VDI**) is a collection of drivers designed to provide applications with a device-independent method of accessing graphically based devices such as monitors, printers, and plotters. Applications which are written to use the **VDI** rather than directly accessing hardware will be compatible with all currently available devices including those which have not yet been developed.

All Atari systems with **TOS** in ROM include a **VDI** screen driver adaptable to each display resolution the system can support. Soft-loaded screen drivers and drivers for other devices are loaded through a **VDI** sub-system called the Graphics Device Operating System (**GDOS**).

The **GDOS** system is disk-loaded as a TSR utility at bootup. It loads device drivers based upon the contents of its configuration file(s).

Applications wishing to use the **GDOS** extensions must verify its presence using the method described later in this chapter. If an application's output will be limited to the screen and no font other than the system font is needed, then the presence of **GDOS** is not mandatory.

VDI Workstations

Every system call made to the **VDI** must include a workstation handle. This handle is a unique integer which identifies the device and current attribute array. Workstation handles are returned by the **VDI** calls **v_opnwk()** or **v_opnvwk()**.

Workstations provide a lookup array of attributes such as line width, text color, clipping state, etc. that are unique to it.

Physical Workstations

Each device must be initialized by opening its physical workstation. Opening a physical workstation causes all drawing and clipping attributes to be reset and the current page (display) to be reset to the default background color. Only one physical workstation may be opened to a single device at any given time.

The screen device's physical workstation is automatically initialized by the **AES** upon bootup. Its physical workstation handle may be obtained from the **AES** call **graf_handle()**.

Devices such as printers and plotters must have their physical workstation opened by the application wishing to utilize them. When opening a physical workstation the application must specify a device ID which identifies the device to open. Device identification codes are assigned as follows:

VDI Device Identification Numbers	
Screen	1–10
Plotters	11–20
Printers	21–30
Metafiles	31–40
Cameras	41–50
Tablets	51–60
Memory	61–70
Other	71–

These values correspond to the value listed in the leftmost column of the user's 'ASSIGN.SYS' file. The following code segment demonstrates opening a physical workstation to the printer device with ID #21. It is important to note that the function assumes that the presence of **GDOS** has been tested for and was verified.

work_in[0] is set to the desired device ID and *work_in[1-9]* are filled in with common defaults for workstation attributes. *work_in[10]* is set to 2 to indicate raster coordinates as explained later in this chapter. The function returns a non-zero value if an error occurred.

```
WORD work_in[11],work_out[57];
WORD handle;

WORD
printer_open( VOID )
{
        WORD i;

        work_in[0] = 21;
        for(i = 1;i < 10; work_in[i++] = 1);
        work_in[10] = 2;

        v_opnwk(work_in,&handle,work_out);

        return (handle == 0);
}
```

Virtual Workstations

Each physical workstation may have multiple virtual workstations opened which allow individual applications to maintain separate workstation attributes. In fact, a single application may open multiple virtual workstations to the same device to manage workstation attributes more efficiently. Opening a virtual workstation does not affect the current contents of the display.

Most **GEM** applications will open a virtual workstation to the current screen device upon initialization. The following code segment illustrates opening a virtual workstation to the display device.

The device identification code for the display device must be specified as **Getrez()** + 2 for all **VDI** features to work correctly. All other parameters are passed the same as the example for

opening a physical workstation except that *handle* must contain the physical workstation handle of the device for which you wish to obtain a virtual workstation handle.

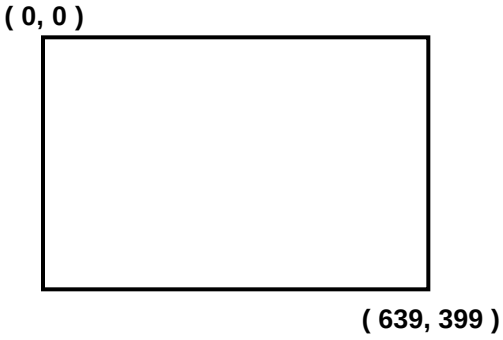
A more programmer-friendly method of opening workstations involves the use of the **VDI_Workstation** structure which is discussed in the reference entry for **V_Opnvwk()**

```
WORD work_in[11],work_out[57];
WORD handle;
WORD wcell, hcell, wbox, hbox;

WORD
screen_open( VOID )
{
        WORD i;

        handle = graf_handle( &wcell, &hcell, &wbox, &hbox);

        work_in[0] = Getrez() + 2;
        for(i = 1;i < 10;work_in[i++] = 1);
        work_in[10] = 2;

        v_opnvwk(work_in, &handle, work_out);

        return (handle == 0);
}
```

Workstation Specifics

Coordinate Systems

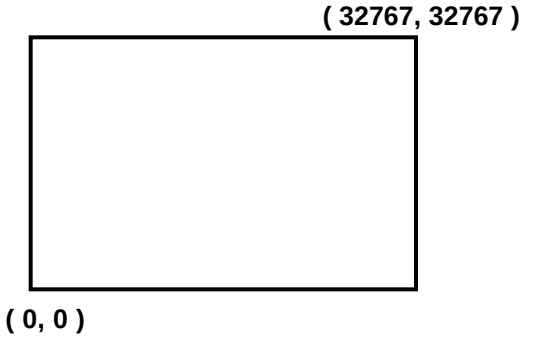
The **VDI** defaults to the usage of Raster Coordinates (RC) which places the origin at the upper-left of the page or display. As an example, the coordinate range for the 1040ST's monochrome graphics mode is shown here:



RC coordinate ranges vary with the device. It is up to the application to interpret and scale the size and position of its output appropriately.

With the addition of **GDOS**, the **VDI** gains the ability to utilize Normalized Device Coordinates (NDC). When using NDC, **GDOS** translates and scales all coordinates to the device as

appropriate. All devices using NDC will have their origin at the lower-left hand corner of the display or page as follows:



Using NDC provides an excellent manner of reducing the overhead of having to internally scale every coordinate, however, applications which depend on the proper aspect ratio for their output should consider managing coordinates internally.

Rendering Graphics

Each **VDI** output function uses attributes set by other related **VDI** functions to determine characteristics such as line width, text face, and color. The following table lists **VDI** attribute calls and the functions they affect.

To output a **VDI** object, set each attribute as desired and then make the appropriate call. For example, to output a line of text in the System font at 9 point colored red, make the following sequence of calls.

```
vst_font( handle, 1 );              /* Select the System Font */
vst_point( handle, 9 );
vst_color( handle, 2 );
v_ftext( handle, 10, 10, “The Atari Compendium” );
```

Generalized Device Primitives

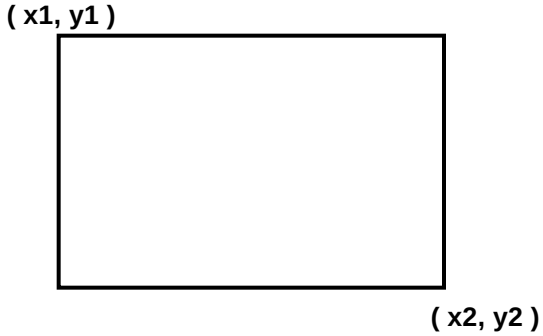
GDP’s (Generalized Device Primitives) are basic drawing components available through the **VDI**. All current device drivers support all GDP’s though specialized drivers may not be able to. *intout[14-24]* may be used to determine the presence of GDP’s. Currently there are 10 supported GDP’s as follows:

#	GDP
1	Bar (Rectangle)
2	Arc
3	Pie Slice
4	Circle
5	Ellipse
6	Elliptical Arc
7	Elliptical Pie
8	Rounded Rectangle
9	Filled Rounded Rectangle
10	Justified Graphics Text

VDI Rectangles

Several **VDI** functions require that a rectangle in **VDI** format be passed to them. **VDI** rectangles are different from **AES** rectangles in the manner in which they are specified.

To correctly define a **VDI** rectangle you must specify two coordinate pairs one representing the upper-left point of the rectangle and the other specifying the lower-right as follows:



The following two functions provide simple conversion between **AES GRECT**s and **VDI** rectangles in an array.

```
VOID
Grect2xy( GRECT *g, short *pxy)
{
       pxy[0] = g.g_x;
       pxy[1] = g.g_y;
       pxy[2] = g.g_x + g.g_w - 1;
       pxy[3] = g.g_y + g.g_h - 1;
}

VOID
Xy2Grect( short *pxy, GRECT *g )
{
       g.g_x = pxy[0];
       g.g_y = pxy[1];
       g.g_w = pxy[2] - pxy[0] + 1;
       g.g_h = pxy[3] - pxy[1] + 1;
}
```

Device Types vs. Required Functions

Not all **VDI** functions are supported by all drivers. The presence of GDP functions may be checked using the information returned in the *intout* array after a **v_opnwk()** call. Other calls may be checked for by entering a test call and comparing returned information with what would be expected.

In addition, each type of driver has a certain number of required functions which must be supported by the device. Each entry in the ***VDI** Function Reference* specifies the support required for a function.

Write Modes

All **VDI** graphics primitives are subject to one of four writing modes set by **vswr_mode()**, with the exception of **vro_cpyfm()** which is passed one of sixteen writing modes.

The following logic tables illustrate the effects of each of the four primary modes. Graphic examples can be found under the reference entry for **vswr_mode()**.

Mode	Logic
Replace	Destination = Source
Transparent	Destination = Source **OR** Destination
XOR	Destination = Source **XOR** Destination
Reverse Transparent	Destination = (**NOT** Source) **AND** Destination

Using Color

The color capabilities of **VDI** devices can be placed into three categories as follows. Determining which category a device falls into is accomplished by examining the return values from **v_opnvwk()**, **v_opnwk()**, and **vq_extnd()**.

Categories	v_opn/v/wk() *work_out[13] { colors }*	vq_extnd() *work_out[5] { lut }*
Monochrome Device[1]	2	0
Palette-Based Device	>= 2	1
True Color Device	> 2	0

[1] Sometimes monochrome devices appear as palette-based devices with two available colors.

Monochrome Devices

Monochrome devices are only capable of displaying one color. Often, monochrome devices are instead represented by palette-based devices with two fixed colors.

Palette-Based Devices

Palette-based devices have a fixed number of colors that may be rendered on screen simultaneously. Each pixel value is used to index into the palette to decide what color to display. For instance, if you change **VDI** color #2 to green, draw a box with that color index, and then change **VDI** color #2 to red, the box will appear first in green and then turn red.

The first 16 **VDI** color registers are used by the operating system and should be avoided. If your application must change them, they should be restored when no longer needed.

True Color Devices

True-color devices allow each pixel to have a unique color value. Rather than palette entries, *colors* (*work_out[13]*) corresponds to the number of available virtual pens. Drawing is accomplished by using these pens, however, unlike using a palette, changing the color of a pen does not change any pixel's color drawn with that pen on the screen.

Whatever color is stored in virtual pen #0 is considered the background color for the purpose of computing write modes.

It is possible for external devices, printers, plotters, etc. to behave as if they were a true-color device.

Color Mapping

Color values are defined in the **VDI** by specifying a red, green, and blue value from 0–1000. The **VDI** will scale the value to the closest color possible. **vq_color()** can be used to determine the actual color that was set.

VDI Raster Forms

The **VDI** handles raster forms using three commands, **vro_cpyfm()**, **vrt_cpyfm()**, and **vr_trnfm()**. **vro_cpyfm()** and **vrt_cpyfm()** are responsible for 'blitting' raster images between memory and a workstation. These functions may also be used to copy images from one location on a workstation to another. 'Blitting' is the process of copying memory from one location to another. Atari computers use the BLiTTER chip (when one is installed) or a software bit blit algorithm to quickly move memory. While these calls are designed to transfer screen memory, if carefully used, they may also be used to transfer other types of memory as well.

vr_trnfm() is responsible for the transformation of images between device-specific and **VDI** standard format, the two raster image formats recognized by the **VDI**. Device-specific format is limited to images in the format of the source device whereas the second is a generic format recommended for transporting images to non-standard displays.

VDI Device-Specific Format

Device-specific format simply mimics the layout of pixels and planes on the source device. When using **vro_cpyfm()** and **vrt_cpyfm()** the source form will be transferred to the destination form in device-specific format[2].

If you intend to save images to disk you should first utilize **vr_trnfm()** to transform the image into a **VDI** standard format so that the image can be successfully ported to any display.

VDI Standard Format

VDI standard format is designed to provide a portable method of specifying raster images which may be displayed on any device. Images stored in **VDI** standard format must be transformed with **vr_trnfm()** before copying them to a workstation.

Images in **VDI** standard format appear in memory in a plane-by-plane fashion. All of the bits for plane #0 appear first followed by the bits for plane #1, and so on for as many planes as exist in the image.

Images may be easily transferred to devices with a higher number of planes by simply inserting empty bytes to account for planes not present in the source image. This method will only work, however, with palette based devices.

Vector Handling

The **VDI** screen driver is also responsible for managing some hardware vectors responsible for keyboard and mouse input. The functions available for altering these vectors are **vex_motv()**, **vex_timv()**, **vex_curv()**, and **vex_butv()**. For further explanation of these calls please see the ***VDI** Function Reference*.

Use of these functions is not recommended with **MultiTOS** as these vectors are global and affect all applications. In addition, results are undefined if two or more non-resident applications utilized these calls at once.

Existing applications which use these calls must have their program flags set to either supervisor or global memory protection. See the ***GEMDOS** Overview* for a discussion of the program flags.

[2]The definitions of **vro_cpyfm()** and **vrt_cpyfm()** allow for the specification of the format of the source and destination form, however, this feature is not currently supported by any version of the operating system. Any call which specifies either the source or destination form to be in device-independent format will fail.

GDOS

The Graphics Device Operating System (**GDOS**) is a disk-based component of the operating system which allows disk-loadable device drivers and additional fonts to be accessible through standard **VDI** calls.

Several versions of Atari **GDOS** have been released in addition to several third-party **GDOS** 'clones'. All of these forms have stayed backward-compatible with **GDOS** 1.0, however it is recommended that programs be written to support newer **GDOS** calls when it can be determined that a more recent release of **GDOS** is present.

Each **VDI** call documented in the ***VDI** Function Reference* specifies if **GDOS** is required, and if so, what type.

Determining the Version of GDOS Present

A non-standard **VDI** call is available to check for the presence of **GDOS**. The following machine-code subroutine will return a longword result in d0 which can be used to determine the variety of **GDOS** present. Beware of older bindings which looked only for the original **GDOS** and returned a 1 or 0 as a result.

```
                .text
_vq_gdos:
                move.l   #-2,d0
                trap     #2
                rts

                .end
```

The longword return value in d0 can be interpreted as follows:

Name	Value	Meaning
GDOS_NONE	-2	No **GDOS** is installed.
—	Any other value.	Original **GDOS** 1.x is installed.
GDOS_FNT	0x5F464E54 '**_FNT**'	**FONTGDOS** is installed.
GDOS_FSM	0x5F46534D '**_FSM**'	**FSM GDOS** or **SpeedoGDOS** is installed. For information on determining the specific variety of outline **GDOS** available, see the description of the 'FSMC' cookie in *Chapter 3:* **BIOS**

FSM GDOS vs. SpeedoGDOS

Since **FSMGDOS** (a QMS/Imagen outline font-based **GDOS**) was never officially released from Atari (though shipped in limited quantity with third-party products), some changes have been made to calls in **SpeedoGDOS** that were never exploited by developers. For that reason, these calls will only be documented in the Speedo-compatible way in the ***VDI** Function Reference*. This does mean, however, that use of these calls will cause your application to fail under the original **FSMGDOS**.

The calls which were affected are **v_getoutline()**, **v_getbitmap_info()**, **v_killoutline()**, and **vqt_get_table()**. In addition, use of the new **SpeedoGDOS** calls **vst_charmap()**, **vqt_trackkern()**, **vqt_pairkern()**, **vqt_fontheader()**, **vst_kern()**, or any of the older calls when used with the **fix31** data type will fail with the older **FSM**.

To determine the type of outline-font **GDOS** installed, look for the '**FSMC**' cookie. The cookie value is a pointer to a longword which contains the character string '**_FSM**' for Imagen-based **FSMGDOS** or '**_SPD**' for Speedo-based **FSMGDOS**.

GDOS 1.x

GDOS 1.0 and the other 1.x versions which followed it was the original **GDOS** developed by Digital Research for Atari. It handled only bitmap fonts and was slow compared to the newer **FONTGDOS** which now replaces it.

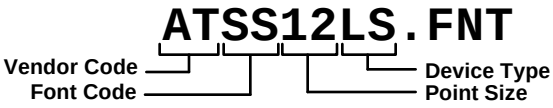
When a **v_opnwk()** call is made with **GDOS** installed, a check is done to see if a driver was assigned to the device ID specified in the 'ASSIGN.SYS' file, and if so, loaded.

All **VDI** calls which specify the returned handle will subsequently be redirected to the driver.

Not all **VDI** functions are available with every driver. Check the 'Availability' heading for each specific function in the ***VDI** Function Reference* for specific availability.

Bitmap Fonts

Bitmap fonts have the ability to be quickly rendered and highly accurate. They do generally require more disk space and a font file must be available for each point size and aspect ratio required. Bitmap fonts follow a special naming convention as follows:



The vendor code is a unique two-letter identifier which specifies the creator of the font. The font code is a two-letter code which abbreviates the font's name. The point size field specifies the point size of the font. The device type is a two-letter abbreviation which should match the aspect ratio of the device as follows:

Device Type	Destination Ratio
None or HI	91x91 (Screen Devices)
CG	91x45 (Screen Devices)
LS	300x300 (Laser Printers, Inkjets)
EP	120x144 (Lo-Res Dot-Matrix Printers)
LB	160x72 (Lo-Res Dot-Matrix Printers)
SP	180x180 (Med-Res Dot-Matrix Printers)
QD	240x216 (Med-Res Dot-Matrix Printers)
NP	360x360 (High-Res Dot-Matrix Printers)

For a driver to recognize a bitmap font it must be listed in the user's 'ASSIGN.SYS' file and be of the correct aspect ratio. No extra fonts are made available to applications until a **vst_load_fonts()** call is made.

FONTGDOS

FONTGDOS is the successor to **GDOS** 1.x. As with the original **GDOS**, **FONTGDOS** supports only bitmap fonts. Its differences are improved driver support, support for bezier curves, improved error handling, and a much quicker response time.

Bezier Curves

FONTGDOS conforms to the **PC-GEM/3** file standard with the inclusion of bezier curve rendering capability with the **v_bez()** and **v_bez_fill()** calls. **v_bez_on()** must be used to allow **FONTGDOS** to allocate the memory necessary for bezier rendering. Likewise **v_bez_off()** should be used before an application exits to free any memory used for bezier calls.

Error Support

When **GDOS** 1.x encountered an error condition, it simply wrote an error message at the top of the display overwriting a portion of the menu bar and display screen. **FONTGDOS** allows an application to disengage this behavior and instead return error codes in a global variable. It is then the applications responsibility to check this variable after calls which may cause an error condition. See the ***VDI** Function Reference* call **vst_error()** for more information.

FSMGDOS

FSMGDOS was developed by Atari in conjunction with QMS/Imagen Corp. to provide Imagen outline fonts which could be displayed at any point size, aspect ratio, or device. It provided all of the improved features of **FONTGDOS** with outline fonts and caching capability. This version of **GDOS** was, however, never officially released. Third-party manufacturers did ship many copies of this **GDOS** to the public. In addition, many developers did update their products to utilize the special features of **FSMGDOS**.

Most **VDI** function calls added with this version of **GDOS** have remained compatible with **SpeedoGDOS**, however, some calls which were never used by developers were changed. This

means that applications written to support **SpeedoGDOS** may not be backwardly compatible. For specific compatibility information, consult the ***VDI** Function Reference*.

SpeedoGDOS

SpeedoGDOS is a new variety of **FSM** which employs outline font technology from Bitstream using Speedo-format outline fonts. In addition, several new calls were added to gain access to internal font information and provide true WYSIWYG (What-You-See-Is-What-You-Get) output.

The fix31 Data Type

SpeedoGDOS optionally allows the use of the **fix31** data type in some calls for parameters and return values. Old bindings designed for the Imagen-based **FSM** will still function properly. Newer bindings may be written to take advantage of this data type.

The **fix31** data type allows for the internal representation and manipulation of floating-point values without the use of a floating-point library. It is a 32-bit value with a 1-bit sign and a 31-bit magnitude. Each value specifies a number in 1/65536 pixels. Examples of this data type follow:

fix31	Floating Point
0x00010000	1.0
0xFFFF0000	-1.0
0x00018000	1.5

Character advances can be simply be added or subtracted to each other using integer arithmetic. To convert a **fix31** unit to an integer (rounding to 0) use the following code:

```
x_integer = (WORD)(x_fix31 >> 16);
```

To convert a **fix31** to an integer and round it to the closest integer use the following code:

```
x_integer = (WORD)((x_fix31 + 32768) >> 16);
```

Use of **fix31** values provides higher character placement accuracy and access to non-integer point sizes. For specific implementation notes, see the ***VDI** Function Reference* entries for **vqt_advance32()**, **v_getbitmap_info()**, **vst_arbpt32()**, and **vst_setsize32()**.

Kerning

SpeedoGDOS outline fonts have the ability to be kerned using two methods. Track kerning is global for an entire font and has three settings, normal, tight, and extra tight. Pair kerning works for individual pair groups of characters. In addition, new pairs may be defined as necessary to produce the desired output.

Kerning is taken into account with **v_ftext()** and **vqt_advance()** only when enabled. Use the calls **vst_kern()**, **vqt_pairkern()**, and **vqt_trackkern()** to access kerning features.

Caching

All **SpeedoGDOS** extent and outline rendering calls are cached for improved performance. Cache files may be loaded or saved to disk as desired to preserve the current state of the cache. In addition, an application might want to flush the cache before doing an output job to a device such as a printer to improve performance with new fonts.

The call **vqt_cachesize()** can be used to estimate the ability of the cache to store data for an unusually large character and prevent memory overflow errors.

Special Effects

The call **vst_scratch()** determines the method used when calculating the size of the special effects buffer. In general an application should not allow the user to use algorithmically generated effects on Speedo fonts. In most cases, special effects are available by simply choosing another font.

The problem is that Speedo fonts may be scaled to any size and **SpeedoGDOS** has no way of predicting the upper-limit on the size of a character to allocate special effects memory. Currently, **SpeedoGDOS** allocates a buffer large enough to hold the largest character possible from the point sizes in the 'ASSIGN.SYS' file and those listed in the 'EXTEND.SYS' file. If your application limits special effects to these sizes then no problems will occur.

If you intend to restrict users to using special effects only with bitmap fonts you may call **vst_scratch()** with a *mode* parameter of 1, memory allocation will be relaxed to only take bitmap fonts into account. You may also specify a *mode* parameter of 2 if you plan to allow no special effects at all. The **vst_scratch()** call must be made prior to calling **vst_load_fonts()**.

Speedo Character Indexes

Speedo fonts contain more characters than the Atari ASCII set can define. Fonts may be re-mapped with a CPX using the **vqt_get_table()** call (this method is not recommended on an application basis as this call affects all applications in the system).

Another method involves the use of a new call, **vst_charmap()**. Calling this function with a *mode* parameter of 0 causes all functions which take character indexes (like **v_ftext()**, **vqt_width()**, etc.) to interpret characters as **WORD**s rather than **BYTE**s and utilize Speedo International Character Encoding rather than ASCII.

The ***Function Reference*** provides two alternate bindings for **v_ftext()** and **v_ftext_offset()** called **v_ftext16()** and **v_ftext_offset16()** which correctly output 16-bit Speedo character text rather than 8-bit ASCII text.

A complete listing of the Bitstream International Character Set is listed in *Appendix G:* ***Speedo Fonts***.

Speedo Font IDs

The function **vqt_name()** is used with all versions of **GDOS** to return a unique integer identifier for each font. Because some bitmap font ID's conflicted with Bitstream outline font ID's, **SpeedoGDOS** versions 4.20 and higher add 5000 to each of the outline font ID's to differentiate them from bitmap fonts.

Device Drivers

Printer and Plotter Drivers

Printer drivers are the most common form of **GDOS** driver available, though some plotter drivers do exist. The ***VDI** Function Reference* can be used to determine if a particular function call is required to be available on a particular device. This does not, however, prohibit the addition of supplementary functions.

Some special printer driver features are available with drivers designed to support them as follows:

Dot-Matrix Printers

Dot-matrix printers with wide carriages can have their print region expanded by passing a custom X and Y resolution for the driver in *ptsin[0]* and *ptsin[1]* respectively prior to the **v_opnwk()** call. In addition, *contrl[1]* should be set to 1 to indicate the presence of the parameters.

SLM804

After a **v_opnwk()** call to an SLM804 driver *contrl[0]* will contain the MSB and *contrl[1]* will contain the LSB of the allocated printer buffer address.

After a **v_updwk()** call, *intout[0]* will contain a printer status code as follows:

Name	Error Code	Meaning
SLM_OK	0x00	No Error
SLM_ERROR	0x02	General Printer Error
SLM_NOTONER	0x03	Toner Empty
SLM_NOPAPER	0x05	Paper Empty

All Printer Drivers

A user-defined printer buffer may be passed to the **v_updwk()** call by specifying the address of the buffer in *intin[0]* and *intin[1]*. In addition, *contrl[3]* must be set to 2 to indicate the new parameters and *contrl[1]* must be set to 1 to instruct the **VDI** to not clear the buffer first.

Camera and Tablet Drivers

As of this writing, no camera or tablet drivers existed for Atari **GEM**. Several functions are reserved to support them which were developed under **PC-GEM**, however, many remain undocumented. Where documentation was available, those calls are included for completeness in the ***VDI** Function Reference*.

The Metafile Driver

'META.SYS' drivers are specially designed drivers which create '.GEM' disk files rather than produce output on a device. When a metafile device is opened, the file 'GEMFILE.GEM' is created in the current **GEMDOS** path. The function **vm_filename()** may be used to change the filename to which the metafile is written to, however, the file 'GEMFILE.GEM' must be deleted by the application.

When a metafile is opened, several defaults relating to the coordinate space and pixel size are set. Each pixel is assigned a default width and height of 85 microns (1 micron = 1/25400 inch). This equates to a default resolution of 300dpi.

The device size is specified where Normalized Device Coordinates (NDC) = Raster Coordinates (RC). The coordinate space of the metafile has (0, 0) in the lower-left corner and (32767, 32767) in the upper-right. This coordinate system may be modified with **vm_coords()**. The size of the actual object space being written to the metafile should also be specified with **vm_pagesize()** so that an application may correctly clip the objects when reading.

After changing coordinate space, values returned by **vq_extnd()** related to pixel width, height and page size will *not* change. Also, font metrics returned by functions such as **vqt_fontinfo()** and **vqt_advance()** will remain based on the default metafile size information. In most cases, text metric information should be embedded based on the workstation metrics of the destination device (such as a screen or printer) anyway.

The metafile is closed when a **v_clswk()** call is issued. Other applications which read metafiles will play back the file by issuing commands in the same order as recorded by the driver. For more information on the metafile format see *Appendix C: **Native File Formats***.

The Memory Driver

'MEMORY.SYS' includes all of the standard **VDI** calls yet works only in memory and is not designed to be output to a device. Normally, the memory driver should be assigned in the user's 'ASSIGN.SYS' file as device number 61. Upon calling **v_opnwk()** to the memory driver, *contrl[1]* should be set to 1 and *ptsin[0]* and *ptsin[1]* should contain the X and Y extent of the memory area. Upon return from the call, *contrl[0]* and *contrl[1]* will contain the high and low **WORD** respectively of the address of the memory device raster. **v_updwk()** clears the raster.

VDI Function Calling Procedure

The **GEM VDI** is accessed through a 68x00 TRAP #2 statement. Prior to the TRAP, register d0 should contain the magic number 0x73 and register d1 should contain a pointer to **VDI** parameter block. An example binding is as follows:

```
        .text
_vdi:
        move.l      #_VDIpb,d1
        move.l      #$73,d0
        trap        #2
        rts
```

The **VDI** parameter block is an array of 5 pointers which each point to a specialized array of **WORD** values which contain input parameters and function return values. Different versions of the **VDI** support different size arrays. The following code contains the 'worst case' sizes for these arrays. Many newer versions of the **VDI** support larger array sizes. You can inquire what the maximum array size that **VDI** supports by examining the *work_out* array after a **v_opnvwk()** or **v_opnwk()**. Larger array sizes allow more points to be passed at a time for drawing functions and longer strings to be passed for text functions. The definition of the **VDI** parameter block follows:

```
            .data

_contrl:    ds.w    12
_intin:     ds.w    128
_ptsin:     ds.w    256
_intout:    ds.w    128
_ptsout:    ds.w    256

_VDIpb:     dc.l    _contrl, _intin, _ptsin
            dc.l    _intout, _ptsout

            .end
```

The *contrl* array contains the opcode and number of parameters being passed the function as follows:

contrl[x]	Contents
0	Function Opcode
1	Number of Input Vertices in *ptsin*
2	Number of Output Vertices in *ptsout*

3	Number of Parameters in *intin*
4	Number of Output Values in *intout*
5	Function Sub-Opcode
6	Workstation Handle
7–11	Function Specific

contrl[0], *contrl[1]*, *contrl[3]*, *contrl[5]* (when necessary), and *contrl[6]* must be filled in by the application. *contrl[2]* and *contrl[4]* are filled in by the **VDI** on exit. *contrl[7-11]* are rarely used, however some functions do rely on them for function-specific parameters.

For specific information on bindings, see the ***VDI*** *Function Reference*.