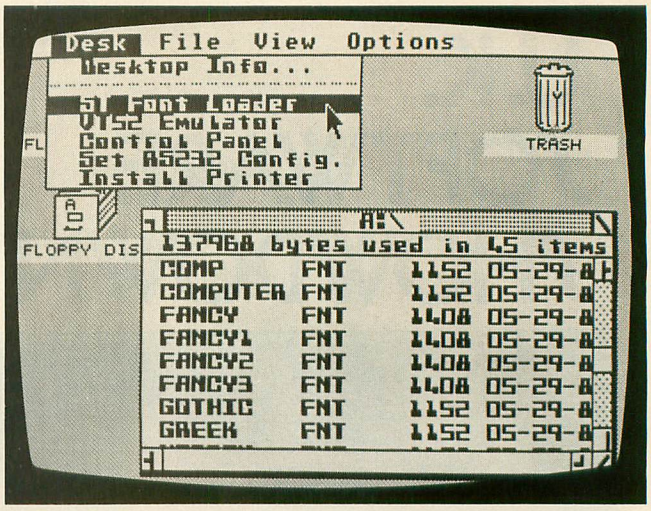
ST FONT LOADER



New character sets in RAM

by JACK POWELL and PATRICK BASS of the **Antic** Staff

In this article, we're going to introduce you to the structure of the 520ST's character fonts and then show you how to change them. Frankly, we're going to have to cheat a bit on this project. The GEM documentation from Digital Research, Inc. refers to a VDI call that loads fonts, but the documentation is incomplete. Until we discover how to properly use this VDI call we'll do a little direct memory manipulation.

We'll create a small C program that takes a standard character set from an 8-bit Atari, reshuffles this character set into the proper font data order for both our 8×8 and 8×16 ST fonts, and then stuffs this data directly into ST font data memory.

Since we're relying on fixed memory locations, we're likely to run into some problems upon future GEM upgrades. But we'll shove ahead, because along the way we'll learn about opening windows, disk access from GEM, file-select boxes, and how to access the Supervisor Mode. We'll also find out how to create a desk accessory, since that's what our program is going to be.

ST'S BUILT-IN FONTS

The Atari ST has three character sets—or "system fonts"—that are loaded into RAM upon power-up.

The characters in each set are identical and are shown in the table on page 63 of the *Atari ST Logo Manual.* The only difference between the sets is the size of the character "cells"—the bit-blocks used to define the characters. For example, in the Atari 8-bit machines, the only cell size is 8×8, or one byte by eight bytes. In the ST there is an 8×16 system font for high resolution, an 8×8 system font for medium and low resolution, and a 6×6 system font which is used for icon labels.

All fonts designed for the GEM system require a certain format and are made up of four parts—Font Header, Character Offset Table, Horizontal Offset Table, and Font Data.

FONT HEADER

Unlike the Atari 8-bit character sets, characters in the ST fonts can each be different widths and of various complexities. The GEM Font Header contains 87 bytes of information describing these features. Most of these instruction bytes are found as the first byte within a word.

Bytes 0-1 of the header are a "face identifier" number. All three system fonts use the number one. Bytes 2-3 are the font "size" in points. The 8×8 system font is a 10-point font.

Bytes 4 through 35 contain a string describing the name of the font. If you look in the memory location for the 8×8 font, you'll find "8×8 system font" followed by a string of zeroes.

The next four bytes contain the lowest ASCII decimal value and the highest ASCII decimal value in the font. In all three system fonts, bytes 36-37 contain zero and bytes 38-39 contain 255.

Bytes 40 through 49 describe the alignment of the characters within their cells. (See *Figure 1.*) Bytes 40-41 hold the "top line" distance, measured from the bottom line. Bytes 42-43 are the "ascent line" distance, 44-45 are the "half line" distance, 46-47 are the "descent line" distance, and 48-49 are the "bottom line" distance. In the 8×8 system font, both the top line and ascent line are 6, the half line

is 4, and the descent as well as the bottom line are 1.

Bytes 50-51 contain the width of the widest character in the font. This is the actual character width and not the cell width. In the 8×8 system font, the widest character is 7. The following bytes, 52-53, hold the widest cell in the font. Naturally enough, the widest cell in the 8×8 system font is 8.

Bytes 54-55 contain the left offset and bytes 56-57 the right offset of the character with the cell.

Bytes 58-59 contain the number of pixels with which to thicken a character. The 8×8 system font, when widened, will thicken by one pixel. You may choose your underline width (in pixels) at bytes 60-61. The 8×8 system font uses 1.

Bytes 62-63 and 64-65 are the "lightening" and the "skewing" masks, respectively. The lightening mask is used to "grey" letters. Skewed letters have an italicized effect. In all three system fonts, both masks are $5555.

Bytes 66-67 contain flag bits. 66 will be zero, and bits 0 through 3 of 67 contain the following flags:

bit 0 set if using default system font
bit 1 set if horizontal offset tables should be used
bit 2 set if byte orientation within a word is high-low
bit 3 set if using a mono-spaced font

If you look at these flags while using the 8×8 system font, bits 0, 2 and 3 will be set.

The next three words in the font header are pointers. Bytes 68 through 71 contain the starting address of the horizontal offset table. Bytes 72 through 74 point to the character offset table, and bytes 76 through 79 point to the font data itself.

The font data may be contained in arrays of varying sizes. In the 8×8 system font, there are 256 cells, each of which are one byte high by eight bytes tall. This can also be described as an array of 256 by 8 bytes. The next four bytes in the font header tell us the width and height of the font data array—or "form".

Bytes 80-81 contain the form width (256 in the 8×8 font) and bytes 82-83 contain the form height (8 in the 8×8 font).

The final four bytes in the font header are a pointer to the address of the next font.

CHARACTER OFFSETS

The Character Offset Table tells the computer where to find each character in the font data by its offset. This permits individual characters of different widths. In the system fonts, each cell is eight bits wide, so a look through the Character Offset Table will reveal a consecutive string of words, each eight more than the last: $0000, $0008, $0010, $0018, $0020, etc. . .

HORIZONTAL OFFSETS

The Horizontal Offset Table gives additional information for added positive or negative spacing of individual characters. It is not used for any of the system fonts and will not be accessed unless its flag bit is set at font header byte 67.

FONT DATA

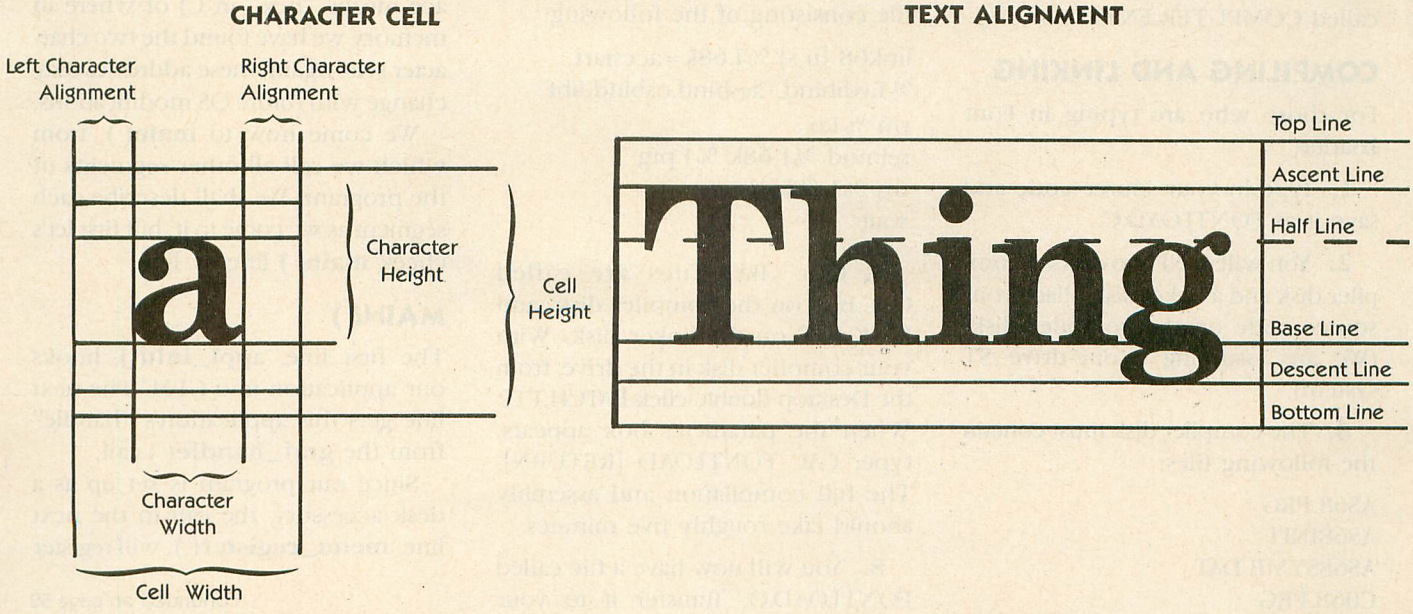
Font Data is the actual bit data that creates the characters in the font. The ST Font Data is arranged quite differently from the Atari 8-bit machines. For example, the data for the letters A and B in the 8-bit machines is arranged so that the first byte, when seen in binary, is the top row of the letter A. The second byte describes the second row of the same letter, the 9th byte is the first row of the letter B, and so on through the set.

But in the ST font data, the first byte would describe the top row of the letter A. The second byte describes the top row of the letter B. The next byte describes the top row of the letter C, and so on.

The actual order of the letters in the ST set can be seen on page 63 of the *Atari ST Logo Manual*. The first byte

continued on next page

Figure 1



of font data is the top row of the "space" or blank. The next byte is the top row of the up arrow, the next is the top row of the down arrow, etc. Since there are 256 characters in the font, the second row of the letter A, for example, will be offset 256 bytes ($100) from the byte representing the top row. This offset value can be found in byte 81—the form width byte—of the font header.

ST FONT LOADER

Until a full-featured, commercial C becomes available, **Antic**'s ST C listings will be written in Alcyon C which is in the Developer's Kit available from Atari for $300. This may seem pricey to you, but remember that the kit includes invaluable documentation for advanced ST programmers, an assembler, an editor, a debugger, and all the proper link files.

If you don't have the Developer's Kit, we've placed the runable object code along with the source code on our 5 1/4-inch monthly disk under the filenames DESK3.ACC, and FONTLOAD.C. Please see the sidebar on page 57 on porting 8-bit Atari files to the ST.

Since the program requires at least one character set file on an ST disk, you'll also need to port (or download) a standard Atari 8-bit character set. Several such character sets are on the ArtDOS disk, available from the Antic Catalog ($10, PD043). Also, this issue's monthly disk includes a sample font called COMPUTER.FNT.

COMPILING AND LINKING

For those who are typing in Font Loader:

1. Type in your source code and save it as FONTLOAD.C.

2. You will need two disks: a compiler disk and a linker disk. Place your source code on the compiler disk. (We are assuming a one-drive ST system).

3. The compiler disk must contain the following files:

AS68.PRG
AS68INIT
AS68SYMB.DAT
C068.PRG
C168.PRG
CP68.PRG
DEFINE.H
GEMDEFS.H
OSBIND.H
OBDEFS.H
BATCH.TTP
RM.PRG
WAIT.PRG
CAC.BAT (User-created file, see below)

4. The linker disk must contain the following files:

ACCSTART.O
AESBIND
VDIBIND
LIBF
OSBIND.O
LINK68.PRG
RELMOD.PRG
BATCH.TTP
RM.PRG
WAIT.PRG
LINK.BAT (User-created file, see below)

5. The BATCH.PRG file on the compiler disk will look for a text file with a .BAT extender consisting of the following:

```
cp68 %1.c %1.i
cO68 %1.i %1.1 %1.2 %1.3 -f
rm %1.i
c168 %1.1 %1.2 %1.s
rm %1.1
rm %1.2
as68 -l -u %1.s
rm %1.s
wait
```

6. The linker disk will need a .BAT file consisting of the following:

```
link68 [u,s] %1.68k=accstart,
%1,vdibind, aesbind,osbind,libf
rm %1.o
relmod %1.68k %1.prg
rm %1.68k
wait
```

7. Our .BAT files are called CAC.BAT on the compiler disk, and LINK.BAT on the linker disk. With your compiler disk in the drive, from the Desktop double-click BATCH.TTP. When the parameter box appears, type: CAC FONTLOAD [RETURN]. The full compilation and assembly should take roughly five minutes.

8. You will now have a file called FONTLOAD.O. Transfer it to your linker disk. Double-click on the linker version of BATCH.TTP and, in the parameter box, type: LINK FONTLOAD [RETURN]. The full linkage should take about five minutes.

9. You now have a file called FONTLOAD.PRG. But, since this program is designed as a desk accessory, it will not run directly from the Desktop. Rename the file DESK3.ACC, then transfer it to a *backup copy* of your power-up disk, along with at least one font file with a .FNT extension. Boot your system with this disk and Font Loader will be found in the drop-down Desk Menu.

Click on ST Font Loader and an Alert Box will announce itself. Click on Proceed and you'll be reminded to put your font disk in drive A. Click on See Disk and the File Selector Box appears showing your fonts. From this point on, everything is self-explanatory.

And, now that you have the program successfully up and running—you do, don't you?—we'll take a thorough look at the listing itself.

PROGRAM TAKE-APART

Right after the remarks and the #include files is an external reference to **gl_apid**, which is the global application ID for the GEM Desktop.

Next come three blocks of variable definitions. Look carefully at the middle two *long* definitions, **char8x8** and **char8x16**. These are the addresses in hexadecimal (the "0x" preface means "hex" in C) of where in memory we have found the two character sets. Again, these addresses may change with future OS modifications.

We come now to **main( )**, from which we call all other segments of the program. We shall describe each segment as we come to it, but first let's check **main( )** line by line.

MAIN( )

The first line, **appl_init( )**, hooks our application into GEM. The next line gets this application's "handle" from the **graf_handle( )** call.

Since our program is set up as a desk accessory, the call in the next line, **menu_register( )**, will register

continued on page 52

continued from page 50

our program with the GEM Desktop by giving GEM the ID of the Desktop **(gl_apid)** and a string containing the title we want in the drop-down Desk Menu. **menu_register( )** returns a number that will uniquely describe this accessory to GEM should we ever want to click on it from the menu. Cunningly, we call this value **menu_id**.

The next call, **wind_get( )**, is a multi-purpose call that here will return the size of the desktop and place the values into **xdesk**, **ydesk**, **wdesk**, and **hdesk**. Before finishing **main( )** we set our window handle to a negative number so we won't mistakenly use another window's number before we are assigned one of our own. To actually perform the program, we jump to **fontable( )**.

OPEN_VWORK( )

Now, we open a virtual workstation, by first filling a **work_in** array with default values, copying the handle number into another variable to pass to GEM, and then performing the **v_opnvwk( )** call, which will return a world of information about what type of terminal we are working on in array **work_out**.

SET_CLIP( )

The **set_clip( )** section defines a rectangle that GEM will not draw outside of. Any line drawn inside this rectangle will appear, but when the line meanders outside, it gets "clipped." We pass this routine the x,y coordinates of the upper left corner of the rectangle and the "w" width and "h" height (in pixels).

OPEN_WINDOW( )

open_window( ), will create and display a window on the desktop. In the first line, **wind_create( )** will create (but NOT display) a window that has its attributes in parameter one, and its maximum size in parameters two, three, four, and five.

This window's only attribute will be the Name line at the top. Other attributes could include a Move-box line, Sliders, Sizing boxes and so forth. The **wind_create( )** call will return a window ID number which we put into **wi_handle**. To actually write the name of the window on the Name line, we use **wind_set( )**.

graf_growbox( ) is optional because all it does is draw the rapidly expanding box outline that precedes the opening of the window itself.

Finally we get to **wind_open( )**, which opens our window onto the screen. In this case we pass the call the handle of the window we want opened, and its size when first opened—which is not necessarily its fullest size.

Last call in this section is **wind_get( )**, which here will return the size of the workspace inside the window we just created and place those values into **xwork** through **hwork**.

DO_REDRAW( )

Whenever our application needs to redraw the screen it goes, logically enough, to **do_redraw**.

GRECT t1, t2 is a "structure" that is defined in the "obdefs.h" #include file. Since we don't have space to describe how structures work in C, remember this is where GEM decides if two rectangles overlap, and how much of each one to re-draw. When GEM is finished drawing our single rectangle on top of everything else on the desktop we perform **do_font( )** to transfer our characters.

DESK ACCESSORIES

We need a pause to discuss how a desk accessory differs from an application program run from a file icon.

Accessories are handled very much like the vertical blank interrupts in the Atari 8-bit computers. That is, sixty times a second, the 6502 processor stops what it is doing and runs off to stuff colors into the hardware registers, or other sundry chores.

In the 520ST, 200 times a second the Dispatcher in the Screen Manager routine checks each of the desk accessories to see if they need service. If service is desired, control is passed to the desk accessory. When a user clicks a menu accessory item, GEM sends a message to the accessory telling it the user has requested the accessory be activated.

The next time the accessory is polled, (200 times a second), the message is received and acted on. The accessory remains active until closed physically or until its job is done and a closure is simulated by software.

FONTABLE( )

Now we come to **fontable( )**, which is called from **main( )**. This is the routine where the program will spend most of its time waiting for a message.

We first start an endless loop, which runs while **TRUE** does not equal false (and that hardly ever happens). **evnt_multi( )**, waits for a multiple combination of "events". Events can include things like key presses, mouse movements, or reception of a message—which is what we are waiting for here.

The list of parameters following the **evnt_multi( )** call are mostly not needed. Here we only pass it the type of event we want (a message—**MU_MESAG**), and the place to put the message received (**msgbuff**).

The next **wind_update( )** tells GEM we are about to update the screen. Then we test if the event was a message and, if not, drop to the last **wind_update( )** which tells GEM to continue drawing anything we stopped during the first **wind_update( )**.

However, if the event *was* the reception of a message, control passes to **switch( )**. The switch. . .case structure performs like multiple IF. . .THENs. There are several possible messages and the type of message is contained in element zero of **msgbuff**, which we pass to **switch( )** for testing.

Of the three possible messages that we are concerned with, **WM_REDRAW** is sent to the accessory to start drawing if needed. **AC_OPEN** tells the accessory the user wants that accessory opened for inspection. **AC_CLOSE** means the user has requested the accessory be cleared from the desktop.

DO_FONT( )

Finally, we reach the routine that performs our work for us. Let's define

what we need to do first. We need to select a font file from the disk, read it in, decode the byte structure and stuff the character images directly into the ST font tables in RAM. Easy, huh?

The first call is **clear_window( )**, which calls a routine below **do_font( )** to erase any information inside the window we just created and opened. The next three lines present the title, wait a little bit, then prompt the user to insert the fonts disk into drive A (see the strings typed in at the top of the listing under the "char" definitions).

Next we come to **fsel_input( )**, which is a completely self-contained call to access the disk directory. You've probably seen the type of box this produces every time you load or save a file in a GEM program.

We need to pass this routine the "path" (which directory we want to see), the "filename" (which will appear in the file slot in the upper right portion of the file select box) and the "button"—actually the address to store the value of the button selected.

First we check to see if the Cancel button has not been selected—**if(button !=0)**.

Graf_mouse(M_OFF, 0x0L) turns off the mouse cursor and **clear_window** clears the window to erase any part of the file selector box left in the window.

Fopen( ) opens the filename gathered from the file selector above (the second parameter is an unused dummy). It will also return a "file handle", or identifier similar to the handle used to identify which screen we are working on. (This is much like the device number on Atari 8-bits). If no file is found, a negative number is returned. The following line checks for this.

Fread( ), will read bytes from a file. We pass it the file handle, the maximum number of characters to read, and where to put the characters it reads—in this case a place called **file_buffer**. **Fread( )** will return a number we call **done**, which is the number of bytes actually read in.

At this point we have read the Font file into memory, so we close the file with **Fclose(file_handle);**.

ENTERING SUPERVISOR MODE

Since we need to access protected low memory where the character font data is stored, we have to switch into Supervisor Mode.

There are at least three ways to enter Supervisor Mode on the 520ST. The process we are going to use here—Bios call #38—is a special case that when called and passed the address of a routine, will enter Supervisor, execute the routine passed, then exit Supervisor and return to the user. So here we call **bios #38**, and pass it the address of the **configure( )** routine which rearranges the stored Atari 8-bit character set into 520ST font format, then pokes that new set into the system font.

After we return from **configure( )**, we turn the mouse shape back on with **graf_mouse(M_ON, 0x0L);**, present a little box that says thanks, close the window we opened previously, draw a quick shrinking box outline, and then delete the window completely.

The last three lines here simulate a "close accessory" message, after which we return to the **event_multi( )** section above.

CLEAR_WINDOW( )

Next follows a short section that contains the instructions to opaque the inside of the window we desire. Otherwise our window work area will appear transparent and we will see the desktop within it. The three **vsf_** calls describe the style and color of the interior—in this case, solid white. The GEM routine, **v_bar( )**, places the fill inside the window described by rectangle "temp".

CONFIGURE( )

We've saved the best for last. **configure( )** is the meat and potatoes of Font Loader.

Before switching fonts, we need to rearrange our old font in two ways. As mentioned earlier, the order of the characters is different, and the order of the character bytes within the whole font array is different. To accomplish this, we use two loops. The outer loop rearranges the character order and the inner loop shuffles the byte order of each character for both the 8 x 8 set and the 8 x 16 hi-res set.

We first declare **point1** and **point2** which will point to the two system character fonts in memory. In the following algorithm, **l** represents the source character we're choosing from and **i** represents the destination character we're working on.

The outer loop will transfer 128 characters from the "source" 8-bit format into the "destination" ST format. Basically, if the character is less than 32, then add 64 to it. If the character is between 32 and 96, then subtract 32 from it to get the index location of the character within the source array.

Since we're dealing with two destination font arrays—8 x 8 and 8 x 16—within the inner loop, we step through all possible 16 scan lines of each high resolution character image then divide by two for the index into the 8 x 8 array. This is done with **(j/2)**, where **j** is an integer.

The following two lines find the offset within both destination fonts in which to place the individual character bytes of the source font. The final two lines actually transfer the data.

If this algorithm seems a little complicated, you might try plugging in some values and following them through on paper.

ST Font Loader was written primarily as a demonstration of GEM programming techniques. We've found that it works on most applications that maintain the GEM menu bar. There is one minor problem. If you want to return to the system font, you must re-boot the computer.

Listing on page 100