Driver Instructions

by Steve Taylor

Well, twist my nipple nuts and send me to Alaska. Here I am again: that makes TWO months in a row. Sheesh, anyone would think I was a sad lifeless git. (Just shut your mouth).

Anyway, welcome to a Fred Guide to Machine Code "Special". Over the next couple of months this column's gonna be dedicated to the technical (ish) documentation for "Driver", the new SAM Wimp environment thingy. Basically, all you lucky Fred readers will be the first people in the entire Universe to learn how to write Driver Applications! Yummy.

Let's start at the beginning. Driver is a program. It's 32k in length with a further 16k of graphics and data. And it runs on any SAM. (With MasterDOS - a SAMDOS version is difficult but I'm looking into it). This fabulous bit of software provides a user interface for other software, in the form of a Windows, Icons, Mouse and Pull-down menus environment.

{ NB. If you don't have a mouse, cursor keys can be used }

WHO ARE YOU CALLING A WIMP?

A WIMP environment is designed to be completely user-friendly. The screen is divided into windows (which can overlap), each of which displays a task, an application program or some data attached to an application. Various options are listed in menus, but are only displayed when the user requests them. Applications or tasks can be reduced to an icon, a small symbol which represents the item. For example, a word processor could be represented by a picture of a pencil.

Now, the user controls the environment using a mouse to direct a pointer around the screen, and the mouse buttons to select items. e.g. clicking on an icon or pulling down a menu.

(I could say that Driver is the SAM version of Windows, but of course that would be breach of copyright so I won't.)

Driver is important in a lot of ways: It pages itself into memory and allocates the stuff using the official guidelines - making sure it's 100% compatible with other things (e.g. MasterBASIC, the dump utility etc). Being 100% machine code, it can co-exist with a BASIC program too.

It also handles the memory management of resident applications. You can have up to 8 fully blown Driver applications sitting in your Coupe simultaneously. Despite the fact that there isn't any real multitasking between them, you can quite happily switch between applications without affecting the state of the files you're editing.

In fact, the applications themselves don't have to worry about the amount of memory your SAM has - Driver allocates memory pages as they are needed, with full use being made of external megs. In theory, you can have files up to 128 16k pages long!!!

WHY DO I WANT TO WRITE A DRIVER APPLICATION?

A good question, mes amis. In a non-commercial situation you can make use of the following advantages:

* A common user-interface, meaning that the user doesn't have to learn lots of new commands from a manual or whatever, and because I've spent a lot of time developing this "front end" it looks much better and is more advanced than that of other SAM applications.

* You don't have to write routines for mundane things like cursor/ pointer movement, menus etc.

* Full compatibility with all SAMs, meaning that there's no need to write separate versions for 256, 512, meg etc.

* Ability to share Driver with other applications at the same point, transferring data between them (using clipboarding)

From a more commercial viewpoint:

* Driver users are going to want more applications than the ones they get with in package.

The main advantage that I found when writing a few applications is the lack of need for routines - everything seems to be provided for you.

HOW DOES IT ALL WORK, STEVE?

I'm glad you asked me that. Try and remember back to last month, when I discussed the use of the frame interrupt for multi-tasking. That principle is the one I've developed for Driver.

Essentially, your application program sits and runs in lower memory, paging the screen and data pages into upper memory. (This arrangement is mainly to allow external RAM to be used. It can only be paged into sections C and D).

At every frame interrupt, your program transfers control to Driver, with does things like scanning the mouse, moving the pointer, flashing the cursor, checking for button presses and so on. If everything's OK control reverts back to the application.

However, if the user has clicked on a window, menu or whatever, Driver kick starts the necessary routines.

Right. I think I'd better start adding some detail, 'cos this is sounding kinda sketchy. At any one time, Driver is running in one of 5 modes:

0 Wimp mode - all window and menu name gadgets are active.
1 Menu mode A - a menu gadget has been clicked, but there is no highlighted "bar"
2 Menu mode B - as above, except that the bar is present.
3 Dialogue mode - Only gadgets in a dialogue box are active.
4 Sleeper mode - No gadgets are active.

Gadgets? Just an umbrella term for the things that can be clicked.

Notice that this splits things up nicely into three sections. And that's just what I'll do.

MODE 0 - WIMP MODE

Your application can be allocated up to 256 windows. Window 0 is ALWAYS the desktop, covering the whole screen.

Windows possess several gadgets for controlling and manipulating them. The available gadgets are as follows:-

Name	Can't be clicked.
Move	Can be clicked and dragged to move the window around.
Size	Can be clicked and dragged to change the size of the window.
Close	Can be clicked to close the window.
X Scroll	A horizontal scroll bar, with left and right arrows and a little square indicating the current scroll position.
Y Scroll	As above, but for vertical scrolling.

Any combination of the above gadgets is possible. The desktop is given, as a default, name and close gadgets only. (You shouldn't change these, except to add scroll bars).

Each window's "type" can be fully specified using 8 bits. D0 is set if you don't want the window area to be cleared before its contents are printed. (For example, if the window contains a large graphic image).

D7 is used to specify whether the window is Selectively-Active or Permanently-Active. A SAW (Selectively-Active-Window) must be clicked to "activate" it, unless it is currently active. A PAW does not. The distinction is made to provide for situations where windows can overlap (the active window is brought to the front). Incidentally, PAWs are always displayed on top of SAWs.

NB. The desktop (window 0) has an additional gadget, a menu bar. It is also permanently active, but is displayed behind all other windows.

The full 8 bit description for each window is as follows:-

D0	Set to avoid window-clearing.
D1	Set for name gadget.
D2	Set for close gadget.
D3	Set for move gadget.
D4	Set for size gadget.
D5	Set for x scroll bar.
D6	Set for y scroll bar.
D7	Set for SAW, reset for PAW.

Driver has a 256 byte long table holding this type for each window, called wintype_tab.

In addition to the gadgets/ type of each window, there are also 256 byte long tables for x coords, y coords, x sizes, y sizes, x scroll positions and y scroll positions.

16 bytes are allocated for each window's name, and there is one further table - 256 bytes holding the order that the windows are displayed on screen. (The entries in this table are the window numbers, with the first byte 0 for the desktop, followed by the next displayed window, then the next etc.)

All these tables are held in the 16k data page I mentioned previously, at the following offsets within the page:

Address (hex)	Name	Comments
1000	wintab	Order of windows.
1100	wintype_tab	Types.
1200	winx_tab	X coords.
1300	winy_tab	Y coords.
1400	winxsize_tab	X sizes.
1500	winysize_tab	Y sizes.
1600	winscrollx_tab	X scroll positions.
1700	winscrolly_tab	Y scroll positions.
1800 ->27FF	winnames	Window names. 16 bytes for each.

Note that the lsb of the address corresponds to the window number.

I should also mention here the co-ordinate system. Driver uses mode 3 (hi-res), but with "fat" co-ordinates. ie. in the x axis, 0 is at the left and 255 is at the far right. Also, in the y axis, 0 is at the TOP with 191 at the bottom.

The text characters used are based on a 6x7 thin pixel grid. In other words, each character uses 3 coords horizontally and 7 coords vertically. This allows 85x26 characters on screen. (I use my own font, not the ROM's)

Finally, the scroll bar tables' entries represent the scroll bar position as a fraction of 255. This means that 0 indicates that the bar is far left (or top), and 255 is far right (or bottom). The scroll bars on screen are scaled to retain this. Your application must translate these values into things like the position through a text file, for example.

MODES 1 AND 2 - MENU MODES

The menus are separated from WIMP mode for simplicity. And they are very simple indeed. On your application desktop you have a menu bar strip thingy that, when clicked, pulls down the appropriate menu. I'll start by describing the menu representation in your application.

* Each menu as stored as the following sequence of bytes:

```
name + carriage return.
number of options
coords (x,y)
option 1, option 2, option 3... etc.
```

The name is stored as ASCII. It can be as long as you like, finishing with a carriage return (0D hex).

The number of options is simply a byte with, um, the number of menu options.

Co-ordinates are ABSOLUTE (using 0,0 at top left of the screen)

* Each option is stored as:

```
type
data
text + carriage return.
```

The type, similar to that for windows, indicates what the option does. A menu option can be on or off (if it's off, it is printed as yellow on white and can't be selected). It can also run a routine when selected, open a sub-menu, toggle a flag or do absolutely nothing at all except split up bits of the menu as a line.

The data is always 8 bytes long: addresses for the routine (or 0 if not applicable), for the sub-menu (0 if N/A) for the flag and finally values for flag on/off.

Note that even if the option doesn't toggle a flag you can still assign one - use 0000 as the flag address to avoid this. If the byte at the flag address matches the "flag on" value a little tick is placed at the left of the option.

Finally, the text is in simple ASCII format, again with 0D hex to terminate. The text is padded out (when it is printed) to be 24 chars long - it mustn't exceed 24 characters.

There is a little bit extra detail I could give you here, connected with sort cut keypresses, but I'll leave that 'til later.

NB. If the "option" is actually a line, just use 0D hex as the text.

One more thing - sub menus are represented in exactly the same way, except that their name is never printed anywhere, so you might as well just use 0D hex.

This is the second part of the Driver Special; last month I discussed how Driver works, and started on modes 0, 1 and 2: WIMP and menu modes.

In a momentary lapse of concentration I failed to mention the exact flags for menu options:

D0	Set if the option is "on".
D1	Set to run a routine on selection.
D2	Set to open a sub-menu.
D3	Set to toggle a flag.
D4	Set to indicate that the option is simply a line.

Note that if you want an option to display a flag, but to run a routine when selected, you simply set both D1 and D3.

Right, now that that's straight, let's continue with

MODE THREE: DIALOGUE MODE

Now, this is good. When you want your application to converse with the user, you can open up a special window (without any of the normal window gadgets) called a dialogue box. Now, you can fill the box with a variety of its own gadgets, in a shorthand form which lets you create quite complex structures without any coding.

In your application, the data for a box looks like this:

0-1	x coord (0-255), y coord for top left of box.
2-3	x size (fat coords), y size.
4	"Reprint screen" flag. One byte, either 0 or 1. A value of 1 will reprint the screen when the box is closed.
5-6	Vector. See below.
7-end	Gadget data.
end	Terminating FFh byte.

The vectored address gets constantly called when the box is open, to make provision for some sort of background tasking using a dialogue box. For example, you could have a box with some sort of meter indicating how much of a file there is left to load/ print, while the thing is loading or printing. In nearly all situations the vector is zero, meaning no call.

Entry to the vector is:

```
A = 1/0 if the box is/isn't being closed.
B = 0 - CANCEL
    1 - OK (for when it is being closed)
```

Return the same registers with the same values, or change the values if you want.

Note that the vector address is bit 15 dependant - see Vectors later on.

The Gadgets.

There are 8 dialogue box gadgets which should provide for every eventuality. The data for each consists of a number 0-7 followed by some parameters: (You just list them end to end; type, parameters, type, parameters.... FFh)

0 - Button: A 32x16 sized box with curved edges and a bit of text inside (up to 9 characters - the text is centred automatically). You can assign a flag to it, so it can be on and off. When it is clicked, a routine is run (see notes below). Examples of buttons I use? OK, CANCEL, CONTINUE etc..

1-2	relative coords
3-4	flag address
5	flag-on value
6-7	address of routine to call when button clicked
8-9	address of text to put inside. (Text ends with FFh)
10-11	address of active-flag. If the byte at that address is 0 the gadget is inactive and cannot be clicked.

1 - Text: Simply a bit of text in the box.

1-2	relative coords
3-4	address of text (ends with FFh)
5-6	address to find colour.

(No click response)

2 - Text box: Similar to using INPUT in a BASIC program. Just put the text in a workspace (see below) and look at the workspace afterwards to see what the user has typed/ changed. A variety of uses, the main one being the entry of file names and the like.

1-2	relative coords
3	workspace no (0-7)
4-5	active-flag address (see above)

3 - Number box: Similar to text boxes, except that it only allows the entry of numbers between 0 and 65535. It needs a temporary workspace, but you set it up with a variable (2 bytes of course) and look at the variable afterwards. Uses? Things like setting page ranges and so on.

1-2	relative coords
3	workspace no (0-7)
4-5	active-flag address (see above)
6-7	variable address

4 - Icon: A rectangular symbol. Make sure it's on a white background like the rest of the dialogue box. I normally use one to represent the purpose of the dialogue, like "attention" and "information" symbols.

1	internal page no. (0 to use application/ driver data)
2-3	address of icon data (D15 = 0 to use application D16 = 1 to use page no. above)
4-5	size (x,y)
6-7	relative coords
7-8	active-flag address (see above)

No click response.

5 - Switch: A bit like buttons, but more like flag menu options. A switch consists of a little square with or without a cross in it (on or off) which you can click, followed by a text label. Like flag options, you can make the on/off values the same, so that clicking merely selects it, or you can make them different so that clicking toggles it on/off.

```
1-2    relative coords
3-4    text address
5-6    flag address
7      flag-on value
8      flag-off value
9-10   active-flag address (see above)
```

6 - User: Provided to let you design your own gadgets. When the user clicks it your routine is called (see notes below). I use it, together with text, icon and box gadgets to make scroll boxes which look complex but are build out of only these four gadgets.

```
1-2    relative coords
3-4    size (x,y)
5-6    routine to call when gadget clicked.
Nothing printed on screen.
```

7 - Box: Simply a rectangle drawn in the dialogue box.

```
1-2    relative coords
3-4    size (x,y)
No click response.
```

Notes
-----

* If a gadget is "inactive" is displayed faded and cannot be selected. You can use this "active-flag" facility to activate, say, a text box when a switch is toggled on, or simply use it to ignore options that aren't available.

* Button routine entered with B = gadget no. (0 being the first in the list). Zero flag set if button is on.

Return A = 0 - no response.
1 - reprint gadgets in box. (If your routine has changed flags)
2 - close box OK
3 - close box CANCEL

* Colour for text gadget: D0-D1 = paper col. (0-3)
D2-D3 = pen col. (0-3)

* User gadget: Click routine entered with A = gadget no. DE = gadget's relative coords within dialogue box. BC = click offset within gadget.

When I use a register pair to pass co-ordinates or size, the lsb is for x, the msb for y. eg. When I use DE to pass coords (as I often do), E holds x coords, D holds y. To convert this to a screen address in upper memory do:

```
SCF
RR D
RR E
```

Simple, eh?

* There are 8 workspaces for text and number boxes, found in the Driver data page at an offset of 3800h. Each is 256 bytes long. You only need to deal directly when using text boxes; you copy the text in before opening the box (with a terminating FFh byte), and copy it back when the box gets closed. In fact, all the text mentioned above uses FFh to terminate.

ADDRESSING

Now, a note on addressing. Your application program runs in lower memory, and pages data into upper memory. However, to distinguish addresses in your application from addresses in Driver and the File Manager (which also run in lower memory), you have to set bit 15 of the address. In effect, you add 8000h to it.

This applies to all the above variable, flag, text and routine addresses except that for icons. It also applies to the menus mentioned last month.

When you want to interface with Driver, you page it into upper memory, and when it wants to interface with your application it pages itself into upper memory. So, although your routine addresses might have D15 set, they get run in lower memory, and you must ensure that Driver is paged into sections C and D when you return.

Also, Driver keeps track of the application stack as well as its own, so you don't have to worry about that at all.

VECTORS

As mentioned previously, Driver runs in parallel with your application, using the frame interrupt. Well, there are occasions when you need to intervene in Driver routines. This is done by using vectors in a similar way to the ROM. Because writing Driver applications is a tad confusing at first, I tried to use things you'd all be familiar with.

For those of you who aren't too sure about vectors, they are simply variables which hold the address of a routine to call in certain situations. If you don't want to use the vector you simply put 0 in the vector.

Again, make sure that all the addresses are "high" (with bit 15 set).

There is a table of application vectors inside Driver, and you set these up when your application is opened. If you want, you can actually set them again to change the values, but they must

all be changed together.

There are 17 vectors in version 1.0 of Driver:

0. Close window: Called when a window close gadget has been clicked. (Not window 0, the application desktop)

Enters A = window no.
Return CY set to keep the window open.

1. Print window: Called during the construction of a window on the screen, after clearing space and drawing the frame. You're supposed to print the window contents and return, whereupon the gadgets are added. There are helpful routines for putting text and graphics in window, more of which next month.

Enters A = window no.
        BC = size. (Remember, lsb (C) = x, msb (B) = y)
        DE = coords.
        B'C' = scroll bar positions.

2. Click: Called when a window interior has been clicked (not a gadget). You're supposed to run the necessary routine(s).

Enters A = window no.
        BC = scroll bar positions.
        DE = click offset within window, after adjusting for the
             name/ menu/ frame.

3. Short-cut key in WIMP mode: Called after the user presses CNTRL with another key (with or without SHIFT/ SYMBOL). This lets you provide short-cuts for things like open/ save/ print/ underline etc.

Enters A = key no. (from SAM keyboard map in p180 of Users
                     Guide)
        +70 for SHIFT
        +140 for SYMBOL.

4. Reserved.

5. Close application: Enters after application desktop closed. You can use it to run a dialogue box, save changes to a file or whatever.

Return A = 0 (close) or 1 (don't)

6. New window: Called when an active SAW is closed (after the close window vector). You can use it to provide the number of the new active SAW. I used it in the file manager to save changes to folders.

Enters A = window no.
Return A = new active SAW.

If the vector is 0 (no routine) the next SAW on the screen is used.

7. Move window: Called when a window is moved. You can use it to adjust the position, or to keep track of it. (Although the position of windows is held in a table, documented last month)

```
Enters  A = window no.
        BC = size.
        DE = new coords.
Return  DE = new coords.
```

8. Resize window: As above, but called after the window's size is changed.

```
Enters  A = window no.
        BC = new size.
        DE = coords.
Return  BC = new size.
```

9. New SAW: Called when a new SAW is activated by clicking it. I used it to do things like save folder attributes.

```
Enters  A = new active SAW.
        B = old active SAW.
```

10. Pointer. This is nice. It gets called every frame in WIMP mode, and lets you change the image used for the pointer depending where it is. For example, IconMaster uses a cross in the editing window and Notepad uses a funny wee "I" thing (I think it's called a caret) in the text window.

```
Enters  A = window pointed to.
        DE = adjusted relative coords. (Just like the click
             vector, although you get negative y values for the
             name/ menu bar)
        BC = window size.

Return  A = page of graphic (0 for application/ Driver data)
        HL = address of new pointer to use. If D7 of H is set,
             Driver data page is used.
```

OR A=H=0 for no change.

11. Scroll up: Called when the scroll-up window gadget is clicked.

```
Enters  A = window no.
        B = current scroll bar position (0-255)
        C = 0 for step, 1 for page.
Return  B = new position.
```

12/13/14. Scroll down/left/right: As above.

15. Mouse scan: Use this to bypass Driver's own scan. You might want to in a graphics package using mode 3 "thin" pixels, although Driver still uses fat ones, so the pointer is still working from 0-255.

```
Enters  DE = pointer coords
Return  DE = new pointer coords.
```

16. Pointer speed. Use this to change to speed of the pointer when key control is in use. (ie. no mouse). The normal speed is 2 pixels per frame in both directions. The vector is called before the coords are changed in WIMP mode. I used it to change speed to 1 pixel per frame when more accuracy was needed in IconMaster and Preferences.

In fact, I stored the window num from the Pointer vector, and used that to judge which speed was needed: slow in the editing window, fast elsewhere.

Enters BC = normal speed (0202h)
Return BC = speed to use.

NB. The key scanning is still done by Driver.

That's the 17 vectors in version 1.0, but I would suggest leaving about 20 zero bytes afterward, to ensure compatibility with later versions.

You should also make sure that Driver is paged into upper memory before returning, and remember that the stack is taken care of.

Last time round I looked at application vectors - today it's the turn of the jump table.

Essentially, vectors transfer control from Driver to the application and the jump table transfers control from the application back to Driver. Simple, yeah? Oh shut your face.

There are about 50 entries in version 1.0, at three byte intervals:

0. JINTERRUPT: Call every frame interrupt.

1. JNMI: Call to reset colours and break back to basic. Also useful for breakpoints when debugging code.

2. JMENU.INIT: Call to initialise menu list with HL = address of list. Remember that all addresses must have D15 set to separate them from the Driver code, although they get accessed in lower memory. See last month for more information on this.

The menu list data consists of the addresses (D15 set) of the menus in order, using 0 to end the list.

9. JVECTOR.INIT: Enter with HL = address of vector list. The list consists of the addresses for the vectors in turn (see last month), with 0 indicating that the vector is not used.

10. JAPPL.INIT: Application initialisation. Enter with HL = application name, DE pointing to the page table and BC = variable table. The first is obvious (end the name with FFh) and gets used for the desktop window. The others are explained later on.

11. JGRAPHIC.INIT: Graphics initialisation. Includes the character set being copied from the Driver data page, the arrow pointer being turned on and the screen being printed.

12. JPRINT.SCREEN: Call to print all the windows in turn, starting with the application desktop. The pointer and cursor are turned off beforehand and then restored afterwards - sleeper mode is also turned off.

13. JPR.PARTSCREEN: Call to print all the windows in turn, starting with the "A"th one displayed. For example, entry with A = 0 has the same effect as JPRINT.SCREEN, A = 1 will start with the first one after the desktop and so on.
Pointer and cursor dealt with as above.

24: JPRINT.WINDOW: Prints window A on the screen. Make sure the pointer and cursor are turned off.

27: JPRINT.SCROLLS: Reprints the scroll bars and move gadget for window A. Useful when something in a window changes and they need to be adjusted. (Of course, each of the three gadgets is only printed is the window has it.)

30: JOPEN.WINDOW: Open window number A at coords DE, with size BC. H holds the type and IX points to its name. (The name ends with FFh and is centred when the window is printed. Even if the window has no name gadget you must assign something!)

Alternatively, enter with A = 0 and a window number will be allocated and returned in A. CY is set if there are no windows available.

The screen display is not changed.

33. JCLOSE.WINDOW: Enter with A = window number to close, including 0 to close the applications. The relevant vector (Close window or close application) will be called by the routine. Screen not updated.

34. JWINDOW.ICON: Prints an icon in window A, trimming it at the right and bottom to fit in. Enter with HL = address of data, DE = relative coords within window, BC = real size, B'C' = size to use after trimming top/left. (Normally the same as BC). Just to confuse you, if HL is >=8000h the Driver data page is used, otherwise the application is used.

If the icon is to be trimmed top/left, DE would be 0 and B'C' would hold the size AFTER trimming. Any further trimming at right/bottom is dealt with.

42. JWINDOW.TEXT: Print some text in window A. DE holds the relative coords, and IX holds the address of the text. As above, trimming is done automatically at right/bottom, but you must enter with C = number of characters to skip (normally 0). To trim at the left, therefore, enter with E = 0 and C = no characters to skip over.

Also enter with A' = paper colour (0-3), B = pen colour.

45. JPOINTER.ON: Enter with A = page, HL = address of pointer data. If A = H = 0, the existing pointer is used.

48. JPOINTER.OFF: Pointer is removed from screen and turned off. The vector JSLEEPER.ON is also called.

51. JCURSOR.ON: Cursor turned on at absolute coords DE. If the cursor is already on, it is removed from the screen and repositioned. However, the mode is not changed (this might mean calling JSLEEPER.OFF to change back to WIMP mode.)

54. JCURSOR.OFF: Cursor removed from screen and turned off.

57. Reserved.

60. Reserved.

63. JSET.MODE: Set Driver mode A.

66. JSLEEPER.ON: Stores current mode and sets mode 4 (sleeper), de-activating all gadgets.

69. JSLEEPER.OFF: Resets previous mode stored by JSLEEPER.ON.

72. JFPAGES: Returns number of free 16k pages in A. If more than 255 are free, 255 is returned.

75. JRESERVE.PAGE: Allocate and reserve page for application. Entries are made in application page table (see later), and the number of pages reserved is increased. If no page is available, the routine returns CY.

78. JRELEASE.PAGE: Releases page at top of page table and decreases page total.

81. JPAGE: Enter with A = logical page number (0-127). Returns physical page numbers in A and B. This is the standard representation of a data page: A represents the HMPR value, and B the LEPR (External page in section C) value. The routine also returns C = lepr, so that on return your application does:

```
OUT (hmpr),A
OUT (C),B
```

84. JRUNCLI: Enter with HL = address of BASIC line data in application. The line is run through the ROM and returns with A = error number or 0 with CY set for an error.

87. JBASIC: Return to BASIC.

90. JDESKTOP: Return to Driver Desktop.

93. Reserved.

96. JGETKEY: Returns key from head of queue in A (ascii code). A = 0 if no key was pressed.

99. JFLUSH: Flush out keyboard queue.

102. JSAVEAS: Opens the "Save as..." dialogue box. Entry is with HL pointing to the file data, which is altered on exit to suit the file details selected by the user. On return, A = 1 (OK) or 0 (CANCEL).

File data consists of 15 bytes:

0/1	Disk number
2	Drive
3	Sub directory number
4	File type (not relevant to "Save as")
5-14	File name, including trailing spaces.

105. JOPENAS: Opens the "Open a file" dialogue box. Entry and exit are as for JSAVEAS. (This is where the file type becomes important).

For opening a file, I would suggest using a default file name with a wildcard, like "*.icn" which would list only those files. This is what Icon Master does.

108. JSCROLLUP: Scroll window A up either a bit (C=0) or a lot (C=1). These correspond to clicking either the up arrow or the space above the scroll bar indicator. The Scroll Up vector is called, and the window(s) redrawn.

111. JSCROLLDOWN: As above, but scrolls down.

114. JSCROLLLEFT: As above.

117. JSCROLLRIGHT: As above.

120. JCUT: Cut BC bytes from HL onto the clipboard. BC must range from 1-256 and HL must be in the application page. CY is returned if the clipboard is full. (It can grow as long as memory allows)

123. JPASTE: Paste BC bytes from the clipboard to HL in the application page. BC must be from 1-256. CY is returned if there is no data left, with BC holding the number of bytes left unpasted.

126. JEMPTYCLIP: Empty clipboard, releasing all its pages, and reset the JCUT pointer to the start.

129. JRESETPASTE: Reset JPASTE pointer to the start of the clipboard.

132. JKEYCONTROL: Enter with A = 1 if the keyboard is to be shared between keyboard control of the pointer (if the user has no mouse) and something else, like entering text. Otherwise A = 0. The user toggles between the two modes using [SYMBOL]-[EDIT].

135. JSELECT.SAW: Select window number A. The affected windows are redrawn on the screen.

138. JBLOCKS: Use HL as the base address for the window gadget blocks. (See Set Variables)

141. JCOPYCHARS: Copy character set from the Driver Data page to a space above the screen. Driver keeps a copy there to speed up printing text (the pointer is also stored above the screen), but some DOS routines might corrupt it.

144. JFREE.PAGE: Returns the top free page available in AB, or CY if none free. No entries are made in any table.

147. JALLOCATE: Allocate page AB in ROM/DOS tables, using value in C. If C is zero, the page is freed. The application page table is not affected.

HOW TO USE THE JUMP TABLE

Using the jump table is simple. Just page Driver into section C and CALL the relevant entry. The numbers given above are offsets - the table is at 8400h in the Driver page, although I'll probably change this at the last minute. Anyway, one of the the set variables (see the section a few pages on) contains the jump table base addess. Note, though, that interrupts are disabled on exit.

The first couple of entries deserve examples. You should be running interrupts in mode 1, and your handler is called at 0038h:

```
maskable.int    PUSH AF
                IN   A,(status)
                RRA
                JP   NC,line.int
                RRA
                RRA
                RRA
                JP   NC,frame.int
maskint.end     POP  AF
                EI
                RET
                .
                .
frame.int       IN   A,(hmpr)
                PUSH AF
                LD   A,(driver.page)
                OUT  (hmpr),A
                CALL jint
                POP  AF
                OUT  (hmpr),A
                POP  AF
                EI
                RET
```

The non-maskable interrupt is called at 0066h:

```
nmi             LD   SP,stack
```

```
CALL driver_in
JP   jnmi
```

This should make it clear that you can run additional routines off the frame interrupt, or a line interrupt. It is possible to run such an interrupt about 20 lines into the screen, and change both the mode and palette. You might want to change the graphics for both the window gadget blocks and the pointer to make them compatible with mode 4. Modes 1 and 2 should be avoided, though. Uses? Art applications, of course.

MEMORY MANAGEMENT

As I mentioned last month, Driver applications run in lower memory, paging data, graphics, the screen and Driver itself into sections C and D. This is to let you use external RAM which can only be paged there.

The external RAM is paged using HMPR and another two ports - LEPR (128 dec) which controls section C, and HEPR (129 dec) for section D. HMPR has D7 set to access the extra memory. So, we can represent any page with two numbers - the HMPR value (0-31 for internal memory, 128 for external) in A and the LEPR value (0-255) in B.

All the extra pages used by your application are held in a table somewhere inside it (PAGETAB). This is 257 bytes long: the first byte represents the number of data pages (0-128), and then the pages are listed using the above format (HMPR for page 0, LEPR for page 0; HMPR for page 1, LEPR for page 1 etc..) JPAGE will return the two values for a logical page number 0-127, although I found it easier to do it myself.

You tell Driver the location of the table via JAPPL.INIT (see above). JRESERVE.PAGE will find a free page (using external memory if possible) and add it into PAGETAB, updating the number of pages. JRELEASE.PAGE does the opposite, freeing the top page.

The problem with this (and about the only programming problem for applications), is that the application data has to be treated in 16k blocks, and the pages won't be concurrent. Not very easy to manipulate, especially for things like word processed documents where you need to keep shuffling memory about. I'll tell you how I got round this drawback with Notepad next month. (Incidentally, you can write documents on Notepad up to 2 MEG BIG!!! - A nice result after having to cope with such a nasty problem!)

There are advantages, though: you don't have to worry about how much memory the machine has, and more than one application can reside and run at the same time. Then the user can cut and paste bits of data between documents without having to load and save. (More on clipboarding next month.)

The one further drawback with the paging is loading and saving. Apart from having to do so using machine code, you can't just use the DOS's usual load and save routines. Instead, you read and write blocks of data at a time. Again, I'll detail this more next month.

One final point: If your application is only going to use 2 data pages (for example, in an art program), you could alway use code like this to page them into either section C or D. Enter with HL = address (8000h-BFFFh for page 0, C000h-FFFFh for page 1):

```
data_in         PUSH AF
                PUSH BC
                PUSH HL
                LD   HL,pagetab+1
                LD   C,(HL)      ; Page 0 hmpr
                INC  HL
                LD   A,(HL)      ; Page 0 lepr
                OUT  (lepr),A
                INC  HL
                LD   B,(HL)      ; Page 1 hmpr
                INC  HL
                LD   A,(HL)
                OUT  (hepr),A    ; Page 1 hepr
                POP  HL
                LD   A,B         ; Use page 0
                BIT  6,H
                JR   Z,di1
                LD   A,C         ; Use page 1
                DEC  A           ; Take 1 off for hmpr
                XOR  C
                AND  31          ; Merge D7, in case of XMEM
                XOR  C
di1             OUT  (hmpr),A    ; Page it
                POP  BC
                POP  AF
                RET
```

You could modify this to use four pages, with all the memory addressed by a single 16-bit word. If the application needs to use more than that, you'll find yourself having to address the memory using 3 bytes. Nasty!

**** IMPORTANT ****

I must stress, once again, that you have to SET BIT 15 on any address that you give Driver to do with your application. The ONLY exception to this is when dealing with icons. Despite this, your application always runs in lower memory.

DRIVER VARIABLES

To keep your application notified, certain Driver variables are copied into a table during every Driver Interrupt. You define the table with JAPPL.INIT:

0. Pointer x coordinate (absolute 0-255)
1. Pointer y coordinate (0-191, 0 is at the top)
2. Mode (0-4)
3. Shift flag. 1 if [SHIFT] is being held, else 0.
4. Mouse button (D0 set when left held, D1 = 0, D2 set when right held.)
5. Pointer flag.
6. Cursor flag.
7. Cursor x coordinate (as for pointer).
8. Cursor y coordinate (as for pointer).
9. Number of windows open, excluding desktop. (0-255)
10. Active SAW number. (1-255, or 0 if none selected)
11. Number of pages reserved by clipboard (0 if none).

After these twelve bytes, leave space for another 10 or so, to ensure compatibility with future versions.

SET ADDRESSES

Inside your application, there are several addresses that must be set aside for specific purposes.

0000 Jumped to when the application is opened for the second time (or later)
0003 Jumped to when the application is opened for the first time only.
0006 Called when Driver is being closed. You must return the address of PAGETAB (with D15 set, of course) in DE.
0009 Reserved for a future "Kickstart" program.

00F0 Name of the application (15 characters max), ending with FFh. This is centred by the Driver Desktop under the application's icon.
0100 Icon. 16x24 fat pixels in the usual format (192 bytes long). Used by both the Driver Desktop and File Manager.

In the first four jumps, the stack is in upper memory, so you need to provide your own. After this, Driver keeps track of the application stack so that every time a vector transfers control from Driver to the application the stack is correct.

One "problem" is the "Driver Closing" jump. Only the currently open application's stack and page table are monitored, so when Driver is closed (and all the applications with it), the stack is in section C and you must return the PAGETAB in DE. You would probably also want to run a dialogue box to save any changes to a file, and your own stack would be needed:

```
0006            JP   driver_closing
 .
 .
 .
driver_closing POP  HL          ; Return address
               LD   SP,stack
               PUSH HL
               CALL close_resave
               CALL driver_in
               LD   DE,pagetab+8000h
               RET
```

As you can see, Driver will revert to its own stack once you return. "driver_in" simply pages Driver into upper memory. An application vector, "close application", is called when the application itself is closed and the fiddling with the stack isn't necessay there.

Incidentally, Driver's page is stored in System Variable 5C97h, and the Driver Data page is at 5C98h. You might like to copy these at the start.

(The idea of the Kickstart thing is that you can put applications in a folder and they will be installed when Driver is loaded.)

FIXED VARIABLES

As I mentioned earlier, at the start of the Driver code there are variables at fixed addresses:

```
0010  PRTOKV.STORE
0012  CMDV.STORE
0014  MTOKV.STORE   Stores for ROM vectors to do with the
keyword.
0016  KEYWORD.FLAG  0 if keyword installed, else 1.
0017  DVAR.OFFSET   Offset of DVAR 0 within the DOS.
0019  Version number of Driver, times 10.
001A  Settings      Offset of preference settings within Driver
code.
001C  ATAB.ADDRESS  Offset of application table within Driver
code.
001E  EDITOR.FLAG   Normally FFh. Use 1 when writing an
application.
001F  BLOCKS        Address of window gadget blocks. To
customise the blocks (for example, when using a different screen
mode) change this with D15 set.
0021  Reserved
0023  Clock vector  See below.
0026  MTask vector  See below.
0029  SEL.APPL      Address of selected application within
application table.
002B  MAX.APPLS     Maximum number of applications (normally
12).
002C  JTAB          Jump table address.
```

MULTI-TASKING

Now, although when I discussed multi-tasking a few months back I mentioned that I'd discounted the idea for Driver, I had a slight change of heart. Although applications are mutually independant when it comes to windows and menus, you can have them running "invisibly" in the background.

This is acheived using the MTask vector mentioned above - this isn't a normal application vector at all. Instead, you specify a page and an address, and after the Driver interrupt has done graphical stuff the vector is called in UPPER MEMORY. And, like the "Driver closing" vector, the stack is left in Driver.

To let more than one application use the vector, another vector is stored three bytes after the vectored address, and is run after the first one. Three bytes after this, the third vector is stored and then run.

Confused? Good. In practice, a "chain" of vectors looks something like this:

```
Driver page,    0026h    18h
                0027h    8010h   ; Page and address of first
vector.

Page 18h,       8010h    JP mtask
                8013h    16h
                8014h    8123h   ; Page and address of second
vector.

Page 16h,       8123h    JP mtask
                8126h    15h
                8127h    8200h   ; Page and address of third
vector.

Page 15h,       8200h    JP mtask
                8203     00h
                8204     0000h   ; No more vectors.
```

Of course, is there aren't any vectors, MTask vector will be 0,0,0.

To set a vector, the following routine (run in lower memory), will be useful:

```
Entry: IX = vector address with D15 set (eg. 8026h for Mtask
vector)
       HL = address of routine with D15 set. (Eg. 8010h)

set.vector      CALL driver_in
                PUSH HL          ; Copy HL into IY
                POP  IY
                LD   A,(IX+0)    ; Store the existing vector
                LD   (IY+0),A
                LD   A,(IX+1)
                LD   (IY+1),A
                LD   A,(IX+2)
                LD   (IY+2),A
                IN   A,(lmpr)    ; Store the new vector
                AND  31
                LD   (IX+0),A
                LD   (IX+1),L
                LD   (IX+2),H
                RET
```

And, at 0010h:

```
0010h          JP   mtask
0013h          DB   0,0,0
```

When you want to disconnect the vector, we have to search through the chain to find it. Then we take the vector from 0013h (or wherever you've put it) and copy it over. Essentially, we've taken the link out of the chain and connected the two loose ends together.

Entry as before.

```
remove.vector  CALL driver_in   ; Driver into upper memory.
rv1            IN   A,(lmpr)
               AND  31
               CP   (IX+0)      ; Check the vector.
               JR   Z,rv.found
               LD   E,(IX+1)    ; Get the address of the wrong one.
               LD   D,(IX+2)
               INC  DE
               INC  DE
               INC  DE
               OUT  (hmpr),A    ; Page it in.
               PUSH DE
               POP  IX          ; Copy it into IX.
               JR   rv1         ; Loop until we find it.
rv.found       INC  HL
               INC  HL
               INC  HL
               LD   A,(HL)      ; Take the link out of the chain.
               LD   (IX+0),A
               INC  HL
               LD   A,(HL)
               LD   (IX+1),A
               INC  HL
               LD   A,(HL)
               LD   (IX+2),A
               RET
```

You might notice that in both routines, the entry conditions include the vector address. This lets the routines also work on ANOTHER vector (gee, I'm too good to you guys).

I mentioned that the MTask vectors are called after the Driver Interrupt has done its graphical stuff. This "graphical stuff" has to done at the start of the interrupt, during the frame flyback to avoid shear. Essentially, the routine looks like this:

```
.
.
Remove pointer from screen
Remove cursor from screen
Call clock vector
Call application graphic vector
Add cursor
Add pointer
.
.
```

The application graphic vector (number 17), which I omitted from last month, is called in WIMP and sleeper modes only, with no entry or return conditions. The clock vector is similar to the MTask vector and can be found at 0023h.

The reasons that I seperated the two are obvious: Firstly, if the screen display is going to be changed, the pointer and cursor must be removed first. Secondly, we want to minimise the time taken by the graphical vectors, or the pointer (and cursor) will be added back on after the screen scan has passed them, making them partly or completely invisible!

I called it the Clock vector because, originally, this was the only use I saw for it; running a clock at the top left of the screen. In fact, such clock application will (hopefully!) be released in one form or another shortly after Driver comes out.

To minimise the time the vector takes, I would suggest doing as much processing as possible in an accompanying MTask routine. The aforementioned clock application uses the MTask vector to "print" the time in a buffer space, which is copied to the screen by the next clock vector. Of course, dumping one rectangular graphic is faster than printing 5 smaller ones.

This month I'm going to describe how to get round the only real problems application programmers will find: Firstly, memory paging.

TOTAL RECALL

Last time round I described how Driver allocates memory in 16k pages, a practice which is pretty obvious given the physical specifications of the Coupe. However, since the user can switch between applications willy nilly, and some applications won't know how much memory they need until it runs out (for example, a word processor), the allocation inevitably leads to memory being broken up and incongruent.

I also mentioned the application page table (APT): How you initialise it, how it stores page numbers, and how to reserve/free pages. I won't bother to repeat all that; you can just look it up.

However, the incongruence of the pages can lead to problems - you've got to be careful not to touch memory in section D (paging in ROM1 could help), and shuffling data about can take ages! This was precisely the nightmare I had with Notepad - the word processor you get in the Driver package. I designed it so that typing automatically inserted characters and deleting automatically pulled all the following text backwards. In addition I wanted to use the Driver clipboard, deleting and inserting large blocks of text. This meant that either

A. I could restrict myself to files 16k long OR

B. I had to find a fast way of copying all the text.

Ugh! The first option was attractive - I didn't have to worry about 19-bit address representation (page/offset), and I could use LDIR and LDDR for simplicity. But it seemed a waste of Driver's facilities, and I wanted a SAM word processor that could handle large files.

Unfortunately, the second option wouldn't work - even in a file 30k long, inserting characters at the start caused noticable delays.

So what did I do? What I always seem to do - I found a compromise that was better than either alternative. And I'm gonna tell you about it...

PAGE BUFFERING

I mentioned that I could handle files up to about 16k long without problems. So, I treated the file in blocks about 16k long - just the way it is represented by the paging mechanism. The sneaky bit is that I left about 256 bytes free at the end of each page, so that the default length of a file block was actually 15.75k long.

So, when the user is typing, the longest block the program has to shuffle (normally) is 15.75k. As this happens, the block size increases until it reaches 16k, then the top 256 bytes are copied into the bottom of the next block, and any overspill from that is copied into the next one... and so on. Obviously, when the user deletes the block size is reduced.

The method means that we need a table of pointers for the top of each block, and some wee routines to adjust any address that points to a non-existant character. We also need routines to insert and delete a block of bytes.

ADJUSTING THE PAGING WHILST COUNTING

This is the first routine I'm gonna give you. We can represent each address with a page number (0-127) in A, and an address offset (8000h-BFFFh) in HL. Put the table of length pointers at an address which is a multiple of 512.

Use the routine after something like INC HL when the paging might need changed - it returns the altered address in AHL, with CY if the physical paging needs to be changed.

```
page.up       PUSH DE
              PUSH HL
              PUSH HL
              LD   L,A
              LD   H,length.tab/512
              ADD  HL,HL
              LD   E,(HL)
              INC  L
              LD   D,(HL)
              POP  HL
              SBC  HL,DE              ; NC if we need to increase
                                      ; the paging.
              POP  HL
              POP  DE
              CCF                     ; Toggle CY.
              RET NC
              LD   HL,8000h
              INC  A
              RET
```

Of course, this can slow things up quite considerably when you're shuffling data about, so it can be a good idea to make a note of the block length when you start, and only update it when you change blocks.

Counting downwards is easier, because the bottom address is always 8000h: Entry and exit as before.

```
page.down     CP   A
              BIT  7,H
              RET  NZ
              DEC  A
              PUSH DE
              LD   L,A
              LD   H,length.tab/512
              ADD  HL,HL
              LD   E,(HL)
              INC  L
              LD   D,(HL)
              LD   L,E
              LD   H,D
              POP  DE
              DEC  HL
              SCF
              RET
```

You might want to vary things a bit to compensate for empty

pages in the middle and so on, but it all depends on the application.

INSERTING A BLOCK

This insert routine is horrendously complicated (heh, heh, heh), considering that all it does it move some data around in the same way that an LDDR command would. You need a buffer 256 bytes long, at a 0-lsb address.

In addition, it can only cope with blocks up to 16383 bytes long - for longer blocks use multiple calls. So why is it so long? (A question I'm often asked. Ahem). Well, I don't really know. It was by far the most difficult thing I had to do for Notepad, and took well over a week to get working.

If the insert is going to cause a page overspill, the block length is reset at 15.75k - of course, the routine has to cope with the possibility that more than one overspill will result. Extra memory is reserved from Driver if necessary, and an error reported if there are any problems ("nomem"). The only other routines needed are:

```
driver_in     Page Driver into section C
data_in       Page data page A into section C.
add_ix_a      Adds A to IX: Result in IX, A unchanged.
```

Take a deep breath...

```
; ENTRY: AHL = address, DE = length of block.

insert.block  LD   (ib.page),A       ; Store for later
              LD   (ib.address),HL
              ADD  A,A
              LD   IX,length.tab     ; Page length table
              CALL add_ix_a
              LD   L,(IX+0)
              LD   H,(IX+1)          ; Page "top" in HL
              ADD  HL,DE             ; Add on block length
              BIT 6,H                ; Page overspill?
              LD   A,(ib.page)
              LD   HL,(ib.address)
              JP   Z,small.insert    ; If not, use a simpler
                                     ; routine.

ib1           CALL ib.check

; If page overspill: Returns CY, HL = 8000h, A incremented, DE =
space needed in next page.
; If no overspill: DE = new end of page, HL = old, A unchanged.

              JR   C,ib1             ; Loop until we've dealt
                                     ; with the overspill
              LD   (ib.countp1),A    ; Store pointers for
                                     ; source/target addresses.
              LD   (ib.countp2),A
              LD   (ib.counta1),HL
              LD   (ib.counta2),DE
              ADD  A,A
              LD   IX,length.tab
              CALL add_ix_a
              LD   (IX+0),E
              LD   (IX+1),D          ; Store new page "top"
              LD   A,(ib.page)
```

```
                LD   C,A                ; Page for start of block
ib2             LD   A,(ib.countp1)
                LD   HL,(ib.counta1)
                CALL data_in
                EX   AF,AF'
                LD   DE,buffer
ib3             EX   AF,AF'
                CP   C                  ; Reached end page
                                        ; (start!)?
                JR   NZ,ib4             ; If not...
                PUSH BC
                PUSH HL
                LD   BC,(ib.address)
                SBC  HL,BC              ; Compare addresses
                POP  HL
                POP  BC
                JR   Z,ib.end           ; Finish main loop if end
ib4             DEC  HL
                BIT  7,H                ; Page overlap?
                JR   NZ,ib7             ; If not, jump
                DEC  A
                PUSH AF
                PUSH IX
                ADD  A,A
                LD   IX,length.tab
                CALL add_ix_a
                LD   L,(IX+0)
                LD   H,(IX+1)           ; Get top address
                LD   (IX+0),0
                LD   (IX+1),BFh         ; Store balance size
                POP  IX
                POP  AF
                CALL data_in
                EX   AF,AF'
                JR   ib3                ; Loop back for empty page
ib7             EX   AF,AF'
                LD   A,(HL)             ; Copy byte to buffer
                DEC  E
                LD   (DE),A
                JR   NZ,ib3             ; Loop for 256 bytes
                EX   AF,AF'
                LD   (ib.countp1),A     ; Store pointers
                LD   (ib.counta1),HL
                LD   A,(ib.countp2)
                LD   DE,(ib.counta2)    ; Target pointers
                LD   HL,buffer
                CALL data_in
                EX   AF,AF'
ib5             DEC  DE
                BIT  7,D                ; Page overlap?
                JR   NZ,ib6             ; If not...
                EX   AF,AF'
                DEC  A                  ; Next page
                LD   DE,BF00h           ; Balance address
                DEC  IX
                DEC  IX
                LD   (IX+0),E
                LD   (IX+1),D           ; Store balance
                CALL data_in
                EX   AF,AF'
                JR   ib5                ; Loop for empty page
ib6             DEC  L
                LD   A,(HL)             ; Copy byte from buffer
                LD   (DE),A
                JR   NZ,ib5             ; Loop for 256 bytes
                EX   AF,AF'
```

```
                LD   (ib.countp2),A    ; Store pointers
                LD   (ib.counta2),DE
                JP   ib2               ; Loop until finished
ib.end          XOR  A                 ; E = bytes left to copy
                SUB  E
                JR   Z,ib.e3           ; If none, jump
                LD   B,A               ; B = 256-E
                LD   A,(ib.countp2)    ; Target pointers
                LD   DE,(ib.counta2)
                LD   HL,buffer
                CALL data_in
                EX   AF,AF'
ib.e1           DEC  DE
                BIT  7,D
                JR   NZ,ib.e2          ; Page overlap?
                EX   AF,AF'
                LD   DE,BF00h
                DEC  IX
                DEC  IX
                LD   (IX+0),E
                LD   (IX+1),D          ; Store balance
                JR   ib.e1
ib.e2           DEC  L
                LD   A,(HL)
                LD   (DE),A            ; Copy from buffer
                DJNZ ib.e1             ; Loop.
                EX   AF,AF'
                LD   (ib.countp2),A

ib.e3           LD   HL,(ib.address)
                LD   BC,BF00h
                CP   A
                SBC  HL,BC
                RET  C
                RET  Z                 ; Ret if start ad<BF01h
                LD   C,L
                LD   B,0
                LD   HL,BF00h
                LD   A,(ib.page)
                CALL data_in
                LD   DE,buffer
                PUSH BC
                LDIR                   ; Copy to buffer
                POP  BC
                LD   A,(ib.countp2)
                CALL data_in
                LD   DE,8000h
                LD   HL,buffer         ; Copy from buffer
                LDIR
                RET

ib.check        PUSH AF
                ADD  A,A
                LD   IX,length.tab
                CALL add_ix_a
                LD   L,(IX+0)
                LD   H,(IX+1)
                ADD  HL,DE
                BIT  6,H
                JR   NZ,ib.c1          ; JR if overspill.
                EX   DE,HL             ; DE = new top
                LD   L,(IX+0)
                LD   H,(IX+1)          ; HL = old top
                POP  AF
                CP   A                 ; Reset CY
                RET
```

```
ib.c1         LD   DE,BF00h
              CP   A
              SBC  HL,DE             ; Subtract BF00h
              EX   DE,HL             ; Put in DE
              LD   HL,8000h
              LD   A,(pagetab)       ; Top page reserved
              LD   B,A
              POP  AF
              INC  A
              CP   B
              RET  C                 ; Return if page exists
              PUSH DE
              CALL driver_in
              CALL jreserve_page     ; Reserve new page
              POP  DE
              JP   C,nomem           ; Error
              LD   A,(pagetab)
              DEC  A                 ; New top page
              LD   HL,8000h
              PUSH AF
              ADD  A,A
              LD   IX,length.tab
              CALL add_ix_a
              LD   (IX+0),L
              LD   (IX+1),H          ; Reset length
              POP  AF
              SCF                    ; Set CY
              RET

small.insert  LD   (ib.page),A
              LD   (ib.address),HL
              ADD  A,A
              LD   IX,length.tab
              CALL add_ix_a
              LD   L,(IX+0)
              LD   H,(IX+1)          ; Top of page
              ADD  HL,DE             ; New top
              PUSH HL
              LD   BC,(ib.address)
              LD   L,(IX+0)
              LD   H,(IX+1)          ; Old top
              CP   A
              SBC  HL,BC             ; Subtract
              JR   Z,si1             ; No bytes to copy
              LD   C,L
              LD   B,H
              LD   L,(IX+0)
              LD   H,(IX+1)          ; Old top
              POP  DE                ; New top
              PUSH DE
              DEC  HL
              DEC  DE
              LD   A,(ib.page)
              CALL data_in
              LDDR                   ; Make space
si1           POP  HL
              LD   (IX+0),L
              LD   (IX+1),H          ; Store new top
              RET
```

DELETING A BLOCK

I was gonna give you the routine, but I've run out of time, and it's pretty easy anyway. Don't bother about intra-page copying - just an LDIR to get rid of stuff inside a page, and adjust the page lengths for other bits.

Right, on with the show. Driver's been selling really well, and all those customers are going to be needing some more luverly applications and utilities. Within a week or two of the official release Darren Clarke had finished writing a game called Mosaic, so it can't be as difficult to understand my notes as I thought! A few other top names in the SAM programming world have expressed serious interest in doing stuff.

We're looking to release some kind of Driver Development Kit shortly, which should consist of a customised version of Comet, Driver and a whole bunch of documentation. I expect you'll have to pay for it, with discounts if you already have Comet/ Driver, but you'd get your money back as soon as you produced something for Revelation to publish.

In addition, there are some more applications (including Alarm Clock, Calendar, Art Grabber, Paintbrush, Cardfile) and other bits and pieces coming out soon(ish) on a Driver Extras disk. Details as soon as I have them.

Okey dokey, back to business. This is the last part of the Driver special (unless I can think of something else for next time), and I'm just gonna cover odd bits and pieces that we've got left over.

CLIPBOARDING

Now I made great claims about clipboarding in the Driver manual. You know, stuff about cutting bits from one file/ application and pasting them in elsewhere. And I stand by it all - it is and will be a great facility, especially when more applications come out. And it's pretty darned easy to use.

Essentially, the clipboard is just some memory, reserved like application data when it is needed and looked after by some Driver routines. There are two ways applications use it: Cutting (putting data on) and pasting (taking data off).

To this end, there are four jump table entries, as I mentioned a few months back:

" 120. JCUT: Cut BC bytes from HL onto the clipboard. BC must range from 1-256 and HL must be in the application page. CY is returned if the clipboard is full. (It can grow as long as memory allows)

123. JPASTE: Paste BC bytes from the clipboard to HL in the application page. BC must be from 1-256. CY is returned if there is no data left, with BC holding the number of bytes left unpasted.

126. JEMPTYCLIP: Empty clipboard, releasing all its pages, and reset the JCUT pointer to the start.

129. JRESETPASTE: Reset JPASTE pointer to the start of the clipboard. "

These simply allow the application to wipe the clipboard clean, add some raw data onto the end and read the raw data back again. To structure things a bit better, we need rules to describe what the raw data represents.

Let's call the things on the clipboard "objects": an object could be a piece of text, a picture or whatever. You can fill up the clipboard with as many objects as there are room for.

Each object is given a four byte header:

0 Object type
1-3 Object length (excluding header) in page/ offset form.

Since the clipboard is a serial store, to read an object later on, you have to read through all the ones before it. It might be the case that your application can only deal with certain data types, in which case you read the headers and skip over any unwanted ones, using the length in the header to guide you.

At the moment, there are only two object types:

0 ASCII text, with Epson control codes. This is the type used by Notepad: the control code sequences begin with CHR$ 1 and end with CHR$ 3, with CHR$ 2 in between the numbers. This lets applications ignore control code strings should they wish.

1 Unmasked graphic block. A rectangular block, with some addition parameters after the header (these bytes are, of course, included in the length in the header):

    0 Mode (0-3)
    1 block width, in bytes. (0-255: the block need not necessarily fit on the screen)
    2 block height, in lines. (0-255)
    3.. graphic data, in raster lines one after the other.

Of course, there can be more than 2 types! It's up to application writers to create new types and document them fully. Subsequent applications might want to be able to deal with these types, and thus the applications could share data. And before you ask, Notepad, alas, cannot cope with graphics. Yet.

Using the four jump table entries is really easy, but the routines vary so much from application to application that it's pretty pointless giving you anything other than hints for your cut/ paste routines. Firstly, you have to do things in blocks of 256 bytes (or less) which means having a buffer space inside the application code. Before pasting, reset the internal paste pointer, and don't forget to empty the clipboard when necessary. Apart from that, it's really just a case of copying data from your appliction data pages to the clipboard and vice versa.

I would suggest using at least three editing options:

Cut (Copies the selection and removes it from the file)
Copy (Copies the selection and does nowt else)
Paste (Paste from the clipboard and then empty it)

Notepad also has a "Multiple paste" option, which doesn't empty the clipboard.

WRITING APPLICATIONS

Finally, after five or so months, we finally get to the bit about actually putting the theory into practice. I used SC_Assembler 256k on a 512k machine to create the source for my Driver applications: this leaves half of the memory untouched and available for Driver code and graphics, and application data. (NB. When I was writing the Driver code itself, I needed all 9 banks in SC_512, so I used an external meg for the File Manager data.)

Anyway, using SC_256 meant installing Driver to page 16, and the Driver data to page 18, without the keyword (the system heap is filled by the assembler). My application object code was assembled to pages 1/2, so I poked EDITOR.FLAG with 1 (the first page number) before initialising Driver. Stuff about running Driver without the keyword is detailed in the manual.

EDITOR.FLAG is at an offset of 1Eh into the Driver code, and on initialisation is copied into the second slot of the Driver application table, after File Manager. (Normally, its value is FFh, meaning "empty").

You can do this for any page you like, but page 1 seems obvious to me. Of course, until you put the application name at 80F0h and icon at 8100h, the Driver Desktop will show garbage, and before assembly any attempt to run the application would result in a bit of a crash! You can design the icon using Iconmaster.

Finally, before even loading Driver, I reserved all the memory I was going to use with OPEN TO 16: CLEAR 25000 or something like that. This customised version of the assembler Basic let me assemble the source, and return directly to Driver to test it. Easy.

So, there's nothing to stop you buying a copy and programming away right now. I expect that LERM and COMET can be altered similarly to have Driver in memory, so you have no excuses!

DRIVER DATA

Right, finally for this month, here are details of the Driver Data page. Some budding artists out there might want to re-draw so of the graphics, or the character set and create alternative driv??.dat files. It can also be useful (if a bit naughty!) for applications to fiddle with data directly. Sprites are in the form described for the pointer routines, and the icons are in the usual raster-line format.

NB. REMEMBER THAT THE DATA PAGE NUMBER IS STORED IN SVAR 5C98h.

Offset	Description
0000 -07FF	Sprites (2k)
0800 -0FFF	Character set (2k)
1000	

```
-27FF   Window data tables (6k)

2800
-37FF   Blocks, icons & other graphics (4k)

3800
-3FFF   Workspaces (2k)
```

SPRITES

```
0000    Arrow
00C6    Hourglass
01CC    Arrowed cross
03F2    Caret
```

CHARACTER SET

The chars (ASCII 32-127) are stored as 16 byte blocks, using 4x8 fat pixels. Driver uses these as 3x7 fat pixels, so the two least significant bits, and the last 2 bytes, should be left blank. The data is NOT the same as with the ROM character set, in that the graphics are stored absolutely, in pen 3 on paper 0. (ROM sets are bit-mapped)

WINDOW DATA TABLES

The LSB of an address in a tables corresponds to the window number 0-255, except in the window order table and the names.

```
1000    Order table.
1100    Types:  D0  Reset to clear the window when drawn.
                D1  Set for name.
                D2  Set for close gadget.
                D3  Set for move gadget.
                D4  Set for size gadget.
                D5  Set for x-scroll bar.
                D6  Set for y-scroll bar.
                D7  Set to make window selectively active.
1200    X coords (fat pixels).
1300    Y coords.
1400    X sizes (fat pixels).
1500    Y sizes.
1600    X-scroll bar positions. (0-255)
1700    Y-scroll bar positions.
1800
-27FF   Names. (4k. 16 bytes per name)
```

OTHER GRAPHICS

```
2800
-29FF   8x8 blocks. Each 32 bytes long.

    Numbered    0 - Move gadget
                1 - Close gadget
                2 - Size gadget
                3 - Up gadget
                4 - Down gadget
                5 - Left gadget
                6 - Right gadget
                7 - Y scroll bar
                8 - X scroll bar
                9 - Scroll bar indicator
                10 - Switch gadget (off)
                11 - Switch gadget (on)
```

12 - Small file icon
13 - Small folder (closed)
14 - Small folder (open)
15 - Small folder (highlighted)

2A00
-2BFF Button icons. 32x16. 256 bytes each.

2A00 - Off
2B00 - On

2C00
-2FFF 16X16 icons. 128 bytes each.

2C00 - Attention (! in a triangle)
2C80 - Disk on blue background
2D00 - Disk on blue (highlighted)
2D80 - Printer
2E00 - Printer (highlighted)
2E80 - Disk on white
2F00 - Info. (i in an octagon)
2F80 - Question (? in octagon)

3000
-35FF 16x24 icons. 192 bytes each.

3000 - File Manager
30C0 - File
3180 - Folder
3240 - Folder (highlighted)
3300 - Bin (empty)
33C0 - Bin (full)
3480 - Bin (empty, highlighted)
3540 - Bin (full, highlighted)

WORKSPACES

Numbered 0-7. Each 256 bytes long at 256 byte intervals from 3800.