Notes on MemMon Internals

Gregg M. Townsend

Department of Computer Science, The University of Arizona

Overall Approach

The programs xmemmon, mmps, mmmeta, mmrt, and mmaed view allocation histories of Icon programs [.ipd148, tr89-30.]. Each program drives a different output device, but most of the code is shared. The shared code, in several files, is linked with a single device-dependent file to make a particular program.

Source Code Organization

For each device-specific program there is a corresponding .c file containing the device-specific source code. Batch mode programs additionally share the file mbatch.c. All programs share these files:

mmain.c	main program, option processing, overall control
mcontrol.c	file interpretation, display control
minput.c	input routines
moutput.c	output routines
mcolors.c	color management
mutils.c	general-purpose utility routines

memmon.h contains global definitions and includes Icon configuration files.

msymbols.def specifies the entries in MemMon's color table. It lists symbol names, labels for use in color specifications, and the corresponding key character on input. This file also specifies, in effect, the screen legend. See the file for more details.

mcolors.def specifies the default color values for the color table. It must be complete with respect to the list of colors in msymbols.def. An alternate color specification may be given at runtime using the MMCOLORS environment variable or the –c option.

Overall Control (mmain.c)

The main program primarily handles option processing and initialization. Most option values are stored in global variables. Because the code is shared, options may be accepted that are not relevant to a particular device.

First, devsetup() is called to initialize device-dependent defaults. Then, options are processed using getopt(). devinit() is called to initialize the output, and /dev/tty is opened for prompting in interactive mode. Following initialization, memmon() is called.

mpause() handles interactive pauses; in batch mode, mpause() initiates frame output.

File Interpretation (mcontrol.c)

The memmon() function contains the program's main loop. Commands are read from is history file and dispatched by a switch statement. None of the cases is very complex.

gcmark() handles the marking phase. It has its own loop and switch statement. Again, most of this is simple code.

Input (minput.c)

getcmd() reads postfix commands from a history file, interpreting numeric arguments and supplying defaults from a table. getshow() and getregion() handle specialized additional information.

Output (moutput.c)

Device-independent output routines are collected in moutput.c. refresh() handles screen refresh, header generation, output scaling, and related matters. paintblk() and paintstr() display a block or a string. mstatus() updates the status line with a prompt or running commentary.

Color Management (mcolors.c)

The global cmap is a color lookup table. With interactive programs, the lookup table is downloaded to the graphics device each time it changes. With batch-mode programs, the color table is maintained locally and used when generating an output frame.

cmap has three sections of size NColors. The first is for the legend and other constant displays. The second (at cmap+Unmarked) is for unmarked blocks, and the third (at cmap+Marked) is for marked blocks during garbage collection. The first two sections are normally full of colors, with all entries in the third set to dark gray.

For example, the word record in the legend is displayed on a block of color C_Record. A record block is displayed in color Unmarked+C_Record, which most of the time is the same as C_Record.

During marking, a marked record is redrawn in color Marked+C_Record. Because this entry is dark gray, the color disappears and marking is apparent on the screen. But just before compaction the Marked colors are copied from the constant region, and the Unmarked colors are set black, to show the active blocks.

After compaction, the color map is restored to its initial state, and strings and blocks are redrawn in Unmarked colors, so everything is then back to normal.

Utilities (mutils.c)

This file contains routines that are useful to MemMon but not specific to it. trim() and chop() remove white-space and break lines into fields; they are used in processing color specification files. litout() sets binary output for drivers writing to tty ports. pexit() aborts a program after reporting file problems.

Device Driver Functions

Interactive device drivers are expected to supply the entry points listed here. For more details about function parameters, etc., see one of the actual drivers.

devsetup()	Set defaults for option processing: –b, –w, –h, –L, –M. Set batchmode=1 if appropriate. Set textsep (see memmon.h).
devinit()	Check –w, –h, etc. and reset if too big. Initialize the output device.
devmap()	Download the color map into the device.
devflood()	Fill the screen with color c.
devpaint()	Draw a block on the memory display, possibly with a different-colored right edge. The location is given as a count of horizontal pixels from the beginning; this is converted to (x,y) coordinates using pre-established global factors.
devtext()	Output a rectangle of one color containing a text string of another color. The rectangle width is strlen(text)+1 characters, with the text indented half a character width. row/column address is given.
devflush()	Flush all output (e.g. for a pause). More output may follow.
devsnap()	Not used, but must be present as a dummy routine.

The Batch-Mode Driver (mbatch.c)

Batch-mode device drivers are linked with additional shared code in mbatch.c. This code emulates a graphics device, saving the necessary information in memory buffers. Then, when mpause() calls devsnap(), mbatch.c writes the buffered data by calling the batch-mode output routines.

mbatch.c contains most the entry points listed above. Batch-mode drivers must supply only:

devsetup()	As above.
devinit()	As above.
batbegin()	Initialize for writing one frame.
battext()	Write a text string; replaces devtext() above.
batmem()	Given an array of pixel values, write a display of the memory regions. Some scaling and other information is available in globals; see actual examples. The memory regions conclude the output for a frame; perform any necessary final processing.
baterm()	Terminate output.

The AED 1024 Driver (mmaed.c)

The AED code actually does not draw rectangles. It draws horizontal lines, then zooms them immensely in the y direction only. Unfortunately, this rules out the display of text.

The reason for this bizarre technique is that the AED is limited by the speed of the serial port. Horizontal lines can be drawn very economically through use of a run-length encoding, about three times faster than standard rectangles. As a side effect, the driver does not need to worry about the ends of lines because wrapping is automatic.

The Raster Tech One/80 Driver (mmrt.c)

There is nothing particularly weird about the Raster Tech driver. Since there is a DMA interface locally, no attempt has been made to minimize the number of bytes sent.

The PostScript Driver (mmps.c)

PostScript files follow Adobe's "Encapsulated PostScript" conventions and consist of a header section followed by one or more output pages. The header section defines a small number of PostScript procedures. Each frame uses standard PostScript commands for setup and then uses the defined procedures for the bulk of the work.

Three PostScript procedures are defined for drawing memory blocks. These procedures maintain a "current address" on the PostScript stack, with the *x* coordinate atop the *y* value. The procedures are:

```
oldy oldx y x n  R  y     x+n+1        % draw rectangle at x, y
    oldy oldx n  A  oldy  oldx+n+1     % draw adjacent rectangle
      oldy oldx  V  oldy  oldx         % draw vertical line at left
```

R draws an *n*-wide block at (*x*, *y*), leaving (*x*+*n*+1, *y*) on the stack. The +1 allows for the usual one-pixel gap between blocks on the display.

A draws an *n*-wide block adjacent to the last block, updating the stack similarly.

V draws a 1-wide block to the *left* of the current location, leaving the stack unchanged. This is designed for drawing string-region separators.

The memory blocks are drawn in order. Colors are switched as needed using a set of C*nn* procedures, one for each unique color.

An additional procedure is used in constructing the header lines:

```
text x y  S  --
```

draws a text string at (*x*, *y*).

The Metafile Driver (mmmeta.c)

Text boxes are output in a straightforward manner. To save time and space, however, the memory regions are written not as series of rectangles but instead as 12-bit color raster images. Each memory region line is a one-unit-high raster image, scaled to almost reach the previous line except for a planned gap.

The X-Windows Driver (xmemmon.c)

xmemmon is easily the most complex driver. It is uses the Xt intrinsics and the Athena widget set, requiring a library no older than Patchlevel 12 of X11 Release 4. It supplies its own main program and does not use mmain.c.

The MemMon code was not designed for the event-driven model used by X-windows. The original structure is preserved by polling for X events when paused and after every 250 calls to devpaint(). This works well except for making menu response a little sluggish during output.

Like other interactive drivers, xmemmon displays blocks on the screen as the history file is read. But because it must be able to refresh the screen in response to an exposure or resize event, it also keeps a copy of the region layouts in its memory, as do the batch-mode drivers.

All facilities are available when using a display with a changeable colormap. If the colormap is not changeable, the only loss is the display of active data after marking. On monochrome screens, textures are used as a substitute for color.

A list of preferred fonts is contained in the driver. The first font from this list that is available and is not too big is used for the legend and header line. The user's default font is used for the buttons. These fonts and many other attributes can be changed using the X Resource Manager.

Adding New Devices

To add a new driver:

- first, learn how to use the device independent part of MemMon
- make a copy of the closest existing driver's C code
- modify that code for the new device's output format
- add new Makefile entries modeled after the closest existing ones
- update the documentation
- build and test

Icon Interpreter (iconx) code

In the Icon interpreter, monitoring code is enabled by default. To disable it, add

```
#define NoMemMon
```

to h/define.h. Small amounts of conditional code are in imain.c (to initialize and terminate memory monitoring) and rmemexp.c. In monitor.h, MM*xxxx* functions are conditionally redefined as null macros so that calls from rmemmgt.c and rmemexp.c do not have to be individually conditionalized. The bulk of the code is in fmonitor.c. The user functions mmout(), mmpause(), and mmshow() are also included in fmonitor.c.

Functions in fmonitor.c are called whenever a block or string is marked or allocated, and at the beginning and end of garbage collection. The fmonitor.c functions do all the actual writing of the history file.

There is extra code in (Icon's) malloc and free to give special treatment to co-expression blocks, and to mark free blocks so they can be identified by a sequential scan of the static region.