XLIB

The "Mode X" graphics library.

Written by: Themie Gouthas.
Additional code contributed by: Matthew MacKenzie and Tore Jahn Bastiansen.
Manual edited by: J. Donavan Stanley.

INTRODUCTION

EDITORS NOTE:

"Do not trust this excuse for a manual when in doubt go to the code." Those words were written by Themie Gouthas in each manual accompanying an XLIB release. I have undertaken this project to provide you the end user with a high-quality, accurate manual to use. I have gone to the code, checked and cross-checked everything and consulted the authors to ensure everything is correct. Just the same I'm only human, so if you spot any errors please let me know.

Throughout this manual function names are in bold italics, variables are in regular italics. This only holds true for those of you who are reading the non-ASCII version of this document. If your favorite word processor isn't included in this release of the manual let me know and I will make very effort to support it. I also plan on making a printed, 3-ring bound versions available for a small fee (Enough to cover the cost of materials and postage) if enough people are interested.

CONTACTING THE AUTHORS:

Themie Gouthas - Internet: egg@dstos3.dsto.goc.au or teg@bart.dsto.gov.au
Matthew MacKenzie - Internet: matm@eng.umd.edu
Tore Bastiansen - Internet: toreba@ifi.uio.no
Donavan Stanley (The editor) - Internet: bbooth@vax.cns.muskingnum.edu (This is a friend who will relay your message to me.)
BBS: The Last Byte BBS (614) 432-3564.

DISCLAIMER

ABOUT XLIB

XLIB is a "user supported freeware" graphics library specifically designed with game programming in mind. It has been released to the public for the benefit of all and is the result of **MANY** hours of or work.

All users **must** comply with the following guidelines:

Leave the code in the public domain.
Do not distribute any modified or incomplete versions of this library.

New contributions and comments are welcome. There will be more releases as the code evolves.

GENERAL FEATURES

- Support for a number of 256 color tweaked graphics mode resolutions
  320x200 320x240 360x200 360x240 376x282 320x400 320x480 360x400 360x480
  360x360 376x308 376x564

- Virtual screens larger than the physical screen (memory permitting) that can be panned at
  pixel resolution in all directions

- A split screen capability for status displays etc.

- Text functions supporting 8x8 and 8x14 ROM fonts and user defined fonts

- Support for page flipping

- Graphics primitives such as line and rectangle drawing functions.

- Bit block manipulation functions

· Please note that some of the resolutions best suit monitors with adjustable vertical height.

MODULES COMPRISING XLIB

XMAIN	Main module containing mode setting code and basic functions
XPOINT	Pixel functions
XRECT	Filled rectangle and VRAM to VRAM block move functions
XPAL	Palette functions
XLINE	Line functions
XTEXT	Text and font functions
XPRINTF	Printf style string output
XPBITMAP	Planar bitmap functions
XPBMCLIP	Clipped planar bitmap functions
XCBITMAP	Compiled bitmap functions using linear bitmaps.
XCOMPPBM	Compiled bitmap functions using planar bitmaps.
XVBITMAP	Video bitmap functions
XMAKEVBM	Support module for video bitmaps
XMOUSE	Mouse functions
XBMTOOLS	Bitmap format conversion tools
XCLIPPBM	Clipped planar bitmap functions. (Uses blits.)
XCIRCLE	Circle Drawing functions.
XDETECT	Hardware detection module
XFILEIO	File I/O functions

XRLETOOL	RLE encoding/decoding functions
XPOLYGON	Convex polygon and triangle functions.
XBEZIER	Bezier curve drawing
XFILL	General purpose flood fill routines.
XVSYNC	Simulated vertical retrace Interrupt module.
XCBITM32	32 Bit compiled bitmaps.

GLOBAL DEFINES (xlib.inc)

Types:

BYTE unsigned char
WORD unsigned int

Available X mode resolutions:

X_MODE_320x200 0
X_MODE_320x240 1
X_MODE_360x200 2
X_MODE_360x240 3
X_MODE_360x282 4
X_MODE_320x400 5
X_MODE_320x480 6
X_MODE_360x400 7
X_MODE_360x480 8
X_MODE_360x360 9
X_MODE_376x308 10
X_MODE_376x564 11

Palette rotation direction:

BACKWARD 0
FORWARD 1

Function return values:

X_MODE_INVALID -1
ERROR 1
OK 0

MODULE XMAIN

The Xmain module is the base module of the XLIB library. It contains the essential functions that initialize and customize the graphic environment.

SOURCES

xmain.asm
xmain.inc
xlib.inc
model.inc

C HEADER FILE

xlib.h

EXPORTED VARIABLES

NOTE: All variables are read only unless otherwise specified. If you modify them manually, the results may be unpredictable.

InGraphics BYTE- Flag indicating that the xlib graphics system is active. Set by function ***x_set_mode***.

CurrXMode WORD - If the xlib graphics system is active, contains the id of the x mode. Set by function ***x_set_mode***. See also: defines (i.e. X_MODE_320x200 ...)

ScrnPhysicalByteWidth WORD - Physical screen width in bytes. Set by function ***x_set_mode***

ScrnPhysicalPixelWidth WORD - Physical screen width in pixels. Set by function ***x_set_mode***

ScrnPhysicalHeight WORD - Physical screen height in pixels. Set by function ***x_set_mode***.

ScrnLogicalByteWidth WORD - Virtual screen width in bytes. Set by function ***x_set_mode***.

ScrnLogicalPixelWidth WORD - Virtual screen width in pixels. Set by function ***x_set_mode***.

ScrnLogicalHeight WORD - Virtual screen height in pixels. Set initially by function ***x_set_mode*** but is updated by functions ***x_set_splitscrn*** and ***x_set_doublebuffer***.

MaxScrollX WORD - Max. X pixel position of physical screen within virtual screen. Set by function ***x_set_mode***.
MaxScrollY WORD - Max. Y pixel position of physical screen within virtual screen. Set initially by function ***x_set_mode*** but is updated by functions ***x_set_splitscrn*** and ***x_set_doublebuffer***.

ErrorValue WORD - Contains error value. General use variable to communicate the error status from several functions. The value in this variable usually is only valid for the last function called that sets it.

SplitScrnOffs WORD - Offset in video ram of split screen. Set by function ***x_set_splitscrn***. The value is only valid if a split screen is active. See also: global variable *SplitScrnActive*.

SplitScrnScanLine WORD - Screen Scan Line the Split Screen starts at initially when set by function ***x_set_splitscrn***. The value is only valid if a split screen is active. See also: global variable *SplitScrnActive*. This variable is not updated by ***x_hide_splitscrn*** or ***x_adjust_splitscrn***.

SplitScrnVisibleHeight WORD - The number of rows of the initial split screen which are currently displayed. Modified by ***x_hide_splitscrn***, ***x_adjust_splitscrn*** and ***x_show_splitscrn***.

Page0_Offs WORD - Offset in video ram of main virtual screen. Initially set by function ***x_set_mode*** but is updated by functions ***x_set_splitscrn*** and ***x_set_doublebuffer***.

Page1_Offs WORD - Offset in video ram of second virtual screen. Set by and only is valid after a call to ***x_set_doublebuffer*** or ***x_triple_buffer.***

Page2_Offs WORD - Offset in video ram of the third virtual screen. Set by and is only valid after a call to ***x_triple_buffer.***

WaitingPageOffs WORD - Offset in video ram of the page waiting to be invisible. Initially set by ***x_triple_buffer*** but is updated by ***x_page_flip***. This variable is only used while triple buffering is active.

HiddenPageOffs WORD - Offset of hidden page. Initially set by function ***x_set_doublebuffer*** but is updated by ***x_page_flip***. This variable is only used while double (or triple) buffering is on.

VisiblePageOffs WORD - Offset of visible page. Initially set by function ***x_set_doublebuffer*** but is updated by ***x_page_flip***. This variable is only used while double (or triple) buffering is on.

NonVisual_Offs WORD - Offset of first byte of non-visual ram, the ram that is available for bitmap storage etc. Set initially by function ***x_set_mode*** but is updated by functions ***x_set_splitscrn*** and ***x_set_doublebuffer***.

VisiblePageIdx WORD - Index number of current visible page. Initially set by function ***x_set_doublebuffer*** but is updated by ***x_page_flip***. This variable is only used while double(or triple) buffering is on.

DoubleBufferActive WORD - Indicates whether double-buffering is on. Set by function ***x_set_doublebuffer***.

TripleBufferActive WORD - Indicates whether triple-buffering is active. Set by function ***x_triple_buffer.***

TopClip, BottomClip, LeftClip RightClip WORD - Defines the clipping rectangle for linear and Video clipped bitmap put functions. Set either manually or by ***x_set_cliprect.***
Note: X coordinates are in bytes as all clip functions clip to byte boundaries.

PhysicalStartPixelX WORD - X pixel Offset of physical (visible) screen relative to the upper left hand corner (0,0) of the virtual screen.

PhysicalStartByteX WORD - X byte Offset of physical (visible) screen relative to the upper left hand corner (0,0) of the virtual screen.

PhysicalStartY WORD - Y pixel Offset of physical (visible) screen relative to the upper left hand corner (0,0) of the virtual screen.

StartAddressFlag WORD - This flag is set if there is a new start address waiting to be set by the vsync handler.

WaitingStartLow, WaitingStartHigh, WaitingPelPan WORD - These are used by the vsync handler only. **Do not modify!**

VsyncPaletteCount WORD - The start index of the video DAC register to be updated during the next vsync. Set by the palette functions.

VsyncPaletteCount WORD - The number of palette entries to be outed during the next vsync. Set by the palette functions.

VsyncPaletteBuffer BYTE[768] - A buffer containing values for the next update of the DAC.

EXPORTED FUNCTIONS

x_set_mode

C Prototype: extern WORD x_set_mode(WORD mode, WORD WidthInPixels);

mode - The required mode as defined by the "Available X Mode resolutions" set of defines in the xlib.h header file.

WidthInPixels - The required virtual screen width.

Returns: The actual width in pixels of the allocated virtual screen

This function initializes the graphics system, setting the appropriate screen resolution and allocating a virtual screen. The virtual screen allocated may not necessarily be of the same size as specified in the *WidthInPixels* parameter as it is rounded down to the nearest multiple of 4.

The function returns the actual width of the allocated virtual screen in pixels if a valid mode was selected otherwise returns X_MODE_INVALID.

Saves virtual screen pixel width in *ScrnLogicalPixelWidth*. Saves virtual screen byte width in *ScrnLogicalByteWidth*. Physical screen dimensions are set in *ScrnPhysicalPixelWidth*. *ScrnPhysicalByteWidth* and *ScrnPhysicalHeight*. Other global variables set are *CurrXMode*, *MaxScrollX*, *MaxScrollY*, *InGraphics*. The variable *SplitScrnScanLine* is also initialized to zero.

See also: "Available X Mode resolutions." and "What is Mode X?"

x_select_default_plane

C Prototype: void x_select_default_plane(BYTE plane);

plane - The plane number you wish to work with.

Enables default Read/Write access to a specified plane

x_set_splitscreen

C Prototype: extern void x_set_splitscreen(WORD line);

line - The starting scan line of the required split screen.

This function activates Mode X split screen and sets starting scan line. The split screen resides on the bottom half of the screen and has a starting address of A000:0000 in video RAM.

It also updates *Page0_Offs* to reflect the existence of the split screen region *MainScrnOffset* is set to the offset of the first pixel beyond the split screen region. Other variable set are *Page1_Offs* which is set to the same value as *Page0_Offs* (See call sequence below), *ScrnLogicalHeight*, *ScrnPhysicalHeight*, *SplitScrnScanLine* and *MaxScrollY*.

This function cannot be called after double buffering has been activated, it will return an error. To configure your graphics environment the sequence of graphics calls is as follows although either or both steps b and c may be omitted:

a) ***x_set_mode***
b) ***x_set_splitscreen***
c) ***x_set_doublebuffer***

Thus when you call this function successfully, double buffering is not active so *Page1_Offs* is set to the same address as *Page0_Offs*.

WARNING: If you use one of the high resolution modes (376x564 as an extreme example) you may not have enough video ram for split screen and double buffering options since VGA video RAM is restricted to 64K.

See Also: "What is a Split Screen?" and "What is double buffering?"

x_set_doublebuffer

C Prototype: extern WORD x_set_doublebuffer(WORD PageHeight);

PageHeight - The height of the two double buffering virtual screens.

Returns: The closest possible height to the requested page height.

This function sets up two double buffering virtual pages. *ErrorValue* is set according to the success or failure of this command.

Other variables set are:

Page1_Offs - Offset of second virtual page

NonVisual_Offs - Offset of first non visible video ram byte

DoubleBufferActive - Flag

PageAddrTable - Table of Double buffering pages start offsets

ScrnLogicalHeight - Logical height of the double buffering pages

MaxScrollY - Max. vertical start address of physical screen within the virtual screen

WARNING: If you use one of the high resolution modes (376x564 as an extreme example) you may not have enough video ram for split screen and double buffering options since VGA video RAM is restricted to 64K.

See Also: "What is double buffering?"

x_triple_buffer

C Prototype: void x_triple_buffer(WORD PageHeight);

PageHeight - The desired height of the virtual screen.

This function behaves like ***x_double_buffer*** but when used with ***x_install_vsync_handler*** you can draw immediately after a page flip. When ***x_page_flip*** is called, *VisiblePageOffs* is set to the page that will be displayed during the next vysnc. Until then, *WaitingPageOffs* will be displayed. You can draw to *HiddenPageOffs*.

See also: "What is triple buffering?"

x_hide_splitscreen

C Prototype: extern void x_hide_splitscreen(void);

This function hides an existing split screen by setting its starting scan line to the last physical screen scan line. *ScreenPhysicalHeight* is adjusted but the *SplitScreenScanLine* is not altered as it is required for restoring the split screen at a later stage.

WARNING: Only to be used if SplitScrnLine has been previously called

Disabled for modes 5-11 (320x400-376x564). The memory for the initial split screen is reserved and the size limitations of these modes means any change in the split screen scan line will encroach on the split screen ram

See Also: "What is a split screen?"

x_show_splitscreen
C Prototype: extern void x_show_splitscreen(void);

Restores split screen start scan line to the initial split screen starting scan line as set by *SplitScrnScanLine*. *ScreenPhysicalHeight* is adjusted.

WARNING: Only to be used if SplitScrnLine has been previously called.

Disabled for modes 5-11 (320x400-376x564). The memory for the initial split screen is reserved and the size limitations of these modes means any change in the split screen scan line will encroach on the split screen ram

x_adjust_splitscreen
C Prototype: extern void x_adjust_splitscreen(WORD line);

line - The scan line at which the split screen is to start.

Sets the split screen start scan line to a new scan line. Valid scan lines are between the initial split screen starting scan line and the last physical screen scan line. *ScreenPhysicalHeight* is also adjusted.

WARNING: Only to be used if SplitScrnLine has been previously called.

Disabled for modes 5-11 (320x400-376x564). The memory for the initial split screen is reserved and the size limitations of these modes means any change in the split screen scan line will encroach on the split screen ram

x_set_start_addr
C Prototype: extern void x_set_start_addr(WORD X,WORD Y);

X, Y - coordinates of top left corner of physical screen within current virtual screen.

Set Mode X non split screen physical start address within current virtual page.

WARNING: X must not exceed (Logical screen width - Physical screen width) i.e. *MaxScrollX,* and *Y* must not exceed (Logical screen height - Physical screen height) i.e. *MaxScrollY*.

x_page_flip

C Prototype: extern void x_page_flip(WORD X,WORD Y);

X,Y - coordinates of top left corner of physical screen within the hidden virtual screen if double buffering is active, or the current virtual screen otherwise.

Sets the physical screen start address within currently hidden virtual page and then flips pages. If double buffering is not active then this function is functionally equivalent to ***x_set_start_addr***.

WARNING: *X* must not exceed (Logical screen width - Physical screen width) i.e. *MaxScrollX*, and *Y* must not exceed (Logical screen height - Physical screen height) i.e. *MaxScrollY*.

x_text_mode

C Prototype: extern void x_text_mode(void);

Disables graphics mode.

x_set_cliprect

C Prototype: extern void x_set_cliprect(WORD left,WORD top,WORD right, WORD bottom);

left, top - X and Y coordinates of the upper left corner of the clipping area.

right, bottom - X and Y coordinates of the lower right corner of the clipping area.

Defines the clipping rectangle for clipping versions of planar and video bitmap puts.

NOTE: Compiled bitmaps cannot be clipped.

MODULE XPOINT

Point functions all MODE X 256 Color resolutions

SOURCES

xpoint.asm
xpoint.inc
xlib.inc
model.inc

C HEADER FILE

xpoint.h

EXPORTED FUNCTIONS

x_put_pix

C Prototype: extern void x_put_pix(WORD X,WORD Y,WORD PageOffset, WORD Color);

X, Y - Coordinates to draw the pixel at.

PageOffset - Virtual page offset to draw on.

Color - The color to use.

Draw a point of specified color.

x_get_pix

C Prototype: extern WORD x_get_pix(WORD X, WORD Y, WORD PageBase);

X, Y - Coordinates of the pixel to get.

PageOffset - Virtual page offset the pixel is located on.

Returns: The color value of the pixel.

Read a pixel from the given coordinates within the given virtual page.

MODULE XRECT

Screen rectangle display and manipulation functions

SOURCES

xrect.asm
xrect.inc
xlib.inc
model.inc

C HEADER FILE

xrect.h

EXPORTED FUNCTIONS

x_rect_pattern

C Prototype: extern void x_rect_pattern(WORD StartX, WORD StartY, WORD EndX, WORD EndY, WORD PageBase,BYTE far *Pattern);

StartX, StartY - Coordinates of upper left hand corner of the rectangle.

EndX, EndY - Coordinates of lower right hand corner of the rectangle.

PageBase - Offset of the virtual screen.

Pattern - Pointer to the user defined pattern (16 bytes).

Mode X rectangle 4x4 pattern fill routine.

Upper left corner of pattern is always aligned to a multiple of 4 row and column. Works on all VGAs. Uses approach of copying the pattern to off-screen display memory, then loading the latches with the pattern for each scan line and filling each scan line four pixels at a time. Fills up to but not including the column at *EndX* and the row at *EndY*. No clipping is performed.

WARNING: The VGA memory locations PATTERN_BUFFER (A000:FFFc) to A000:FFFF are reserved for the pattern buffer

See Also: Doctor Dobbs Journal references.

x_rect_pattern_clipped

C Prototype: extern void x_rect_pattern_clipped(WORD StartX, WORD StartY, WORD EndX,
WORD EndY, WORD PageBase,
BYTE far *Pattern);

StartX, StartY - Coordinates of upper left hand corner of the rectangle.

EndX, EndY - Coordinates of lower right hand corner of the rectangle.

PageBase - Offset of the virtual screen.

Pattern - Pointer to the user defined pattern (16 bytes).

Mode X rectangle 4x4 pattern fill routine.

Upper left corner of pattern is always aligned to a multiple of 4 row and column. Works on all VGAs. Uses approach of copying the pattern to off-screen display memory, then loading the latches with the pattern for each scan line and filling each scan line four pixels at a time. Fills up to but not including the column at *EndX* and the row at *EndY*. Clipping is performed.

WARNING: The VGA memory locations PATTERN_BUFFER (A000:FFFc) to A000:FFFF are reserved for the pattern buffer

See Also: Doctor Dobbs Journal references.

x_rect_fill

C Prototype: extern void x_rect_fill(WORD StartX, WORD StartY, WORD EndX,
WORD EndY, WORD PageBase, WORD color);

StartX, StartY - Coordinates of upper left hand corner of the rectangle.

EndX, EndY - Coordinates of lower right hand corner of the rectangle.

PageBase - Offset of the virtual screen.

Color -color to use for fill

Mode X rectangle solid color fill routine.

Based on code originally published in DDJ Magazine by M. Abrash

See Also: Doctor Dobbs Journal references.

x_rect_fill_clipped

C Prototype: extern void x_rect_fill_clipped(WORD StartX, WORD StartY, WORD EndX, WORD EndY, WORD PageBase, WORD color);

StartX, StartY - Coordinates of upper left hand corner of the rectangle.

EndX, EndY - Coordinates of lower right hand corner of the rectangle.

PageBase - Offset of the virtual screen.

Color - The color to use for fill.

Mode X rectangle solid color fill (With clipping) routine.

Based on code originally published in DDJ Magazine by M. Abrash

See Also: Doctor Dobbs Journal references.

x_cp_vid_rect

C Prototype: extern void x_cp_vid_rect(WORD SourceStartX, WORD SourceStartY,
WORD SourceEndX, WORD SourceEndY,
WORD DestStartX, WORD DestStartY,
WORD SourcePageBase, WORD
DestPageBase,
WORD SourceBitmapWidth, WORD
DestBitmapWidth);

StartX, StartY- Coordinates of the upper left hand corner of the source rectangle.

EndX, EndY - Coordinates of the lower right hand corner of the source rectangle.

DestStartX, DestStartY - Coordinates of the rectangle's destination.

SourcePageBase - Source rectangle page offset.

DestPageBase - Destination rectangle's page offset.

SourceBitmapWidth - The width of bitmap within the source virtual screen containing the source rectangle

DestBitmapWidth - The width of bitmap within the dest. virtual screen containing the destination rectangle

Mode X display memory to display memory copy routine. Left edge of source rectangle modulo 4 must equal left edge of destination rectangle modulo 4. Works on all VGAs. Uses approach of reading 4 pixels at a time from the source into the latches, then writing the latches to the destination. Copies up to but not including the column at *SrcEndX* and the row at *SrcEndY*. No clipping is performed. **Results are not guaranteed if the source and destination overlap**.

Based on code originally published in DDJ Magazine by M. Abrash

See Also: Doctor Dobbs Journal references.

x_shift_rect

C Prototype: extern void x_shift_rect (WORD SrcLeft, WORD SrcTop, WORD SrcRight, WORD SrcBottom, WORD DestLeft, WORD DestTop, WORD ScreenOffs);

SrcLeft, SrcTop - Coordinates of the upper left hand corner of the rectangle.

SrcRight, SrcBottom - Coordinates of the lower right hand corner of the rectangle.

DestLeft, DestTop - Coordinates of the upper left corner of the destination.

ScreenOffs - Offset of the virtual screen.

This function copies a rectangle of VRAM onto another area of VRAM, even if the destination overlaps with the source. It is designed for scrolling text up and down, and for moving large areas of screens around in tiling systems. It rounds all horizontal coordinates to the nearest byte (4-column chunk) for the sake of speed. This means that it can NOT perform smooth horizontal scrolling. For that, either scroll the whole screen (minus the split screen), or copy smaller areas through system memory using the functions in the XPBITMAP module.

SrcRight is rounded up, and the left edges are rounded down, to ensure that the pixels pointed to by the arguments are inside the rectangle. That is, *SrcRight* is treated as (*SrcRight*+3) >> 2, and *SrcLeft* as *SrcLeft* >> 2.

NOTE: The width of the rectangle in bytes (width in pixels / 4) cannot exceed 255.

MODULE XPAL

Palette functions for VGA 256 color modes.

All the functions in this module operate on two variations of the palette buffer, the raw and annotated buffers.

All those functions ending in "raw" operate on the following palette structure:
BYTE:r0,g0,b0,r1,g1,b1,...rn,gn,bn No reference to the starting color index or number of colors stored is contained in the structure.

All those functions ending in "struc" operate on the following palette structure:
BYTE:c,BYTE:n,BYTE:r0,g0,b0,r1,g1,b1,...rn,gn,bn where *c* is the starting color and *n* is the number of colors stored

WARNING: There is no validity checking in these functions. The onus is on the user to supply valid parameters to the functions.

SOURCES

xpal.asm
xpal.inc
xlib.inc
model.inc

C HEADER FILE

xpal.h

EXPORTED FUNCTIONS

x_get_pal_raw

C Prototype: extern void x_get_pal_raw(BYTE far * pal,WORD num_clrs, WORD index);

pal - Pointer to a buffer to receive the raw palette.

num_clrs - The number of colors to get.

index - Starting color number to get.

Read DAC palette into raw buffer with interrupts disabled i.e. BYTE r1,g1,b1,r1,g2,b2...rn,gn,bn

WARNING: Memory for the palette buffers must all be pre-allocated.

x_get_pal_struc

C Prototype: extern void x_get_pal_struc(BYTE far *pal, WORD num_clrs, WORD index);

pal - Pointer to a buffer to receive the palette structure.

num_clrs - The number of colors to get.

index - The starting color number to get.

Read DAC palette into annotated type buffer with interrupts disabled i.e. BYTE colors to skip, BYTE colors to set, r1,g1,b1,r1,g2,b2...rn,gn,bn

WARNING: Memory for the palette buffers must all be pre-allocated.

x_put_pal_raw

C Prototype: extern void x_put_pal_raw(BYTE far *pal, WORD num_clrs, WORD index);

pal - Pointer to a buffer containing the raw palette.

num_clrs - The number of colors to put.

index - Starting color number to put.

Write DAC palette from raw buffer with interrupts disabled i.e. BYTE r1,g1,b1,r1,g2,b2...rn,gn,bn

x_put_pal_struc
C Prototype: extern void x_put_pal_struc(BYTE far * pal);

pal - Pointer to a buffer containing the palette structure.

Write DAC palette from annotated type buffer with interrupts disabled i.e. BYTE colors to skip, BYTE colors to set, r1,g1,b1,r1,g2,b2...rn,gn,bn

x_set_rgb
C Prototype: extern x_set_rgb(BYTE color,BYTE red_c,BYTE green_c, BYTE blue_c);

color - The color number to modify.

red_c - The red component for this color.

green_c - The green component for this color.

blue_c - The blue component for this color.

Set the RGB components of a VGA color

x_rot_pal_struc
C Prototype: extern void x_rot_pal_struc(BYTE far * pal, WORD direction);

pal - Pointer to the palette structure to rotate.

direction - The direction to rotate the palette.

Rotate annotated palette buffer entries. Direction 0 = backward, 1 = forward.

x_rot_pal_raw
C Prototype: extern x_rot_pal_raw(BYTE far * pal, WORD direction, WORD num_colrs);

pal - Pointer to the raw palette buffer to rotate.

direction - The direction to rotate the palette.

num_colrs - The number of colors in the buffer.

Rotate a raw palette buffer. Direction 0 = backward, 1 = forward.

x_put_contrast_pal_struc

C Prototype: extern void x_put_contrast_pal_struc(BYTE far * pal, BYTE intensity);

pal - A pointer to the palette structure to modify.

intensity - Number of units to decrement the palette.

Write DAC palette from annotated type buffer with specified intensity adjustment (i.e. palette entries are decremented where possible by "intensity" units).
Designed for fading in or out a palette without using an intermediate working palette buffer! (Slow but memory efficient ... OK for small pal structs}

x_transpose_pal_struc

C Prototype: extern void x_transpose_pal_struc(BYTE far * pal, WORD StartColor);

pal - Pointer to the palette structure to modify.

StartColor - Starting color index.

Write DAC palette from annotated type buffer with interrupts disabled starting at a new palette index.

x_cpcontrast_pal_struc

C Prototype: extern WORD x_cpcontrast_pal_struc(BYTE far *src_pal, BYTE far *dest_pal,
BYTE Intensity);

src_pal - Pointer to the source palette structure.

dest_pal - Pointer to the destination palette structure.

Intensity - Number of units to decrement the palette.

Copy one annotated palette buffer to another making the intensity adjustment. Used in fading in and out fast and smoothly.

MODULE XLINE

Line Drawing functions.

SOURCES

xline.asm
xline.inc
xlib.inc
model.inc

C HEADER FILE

xline.h

EXPORTED FUNCTIONS

x_line

C Prototype: extern void x_line(WORD x0,WORD y0,WORD x1,WORD y1,
WORD color,WORD PageBase);

x0 - Starting X coordinate.

y0 - Starting Y coordinate.

x1 - Ending X coordinate

y1 - Ending Y coordinate.

color - The color to use for the line.

PageBase - The page offset on which to draw the line.

Draw a line with the specified end points in the specified page.
No Clipping is performed.

MODULE XTEXT

Mode X text display functions.

SOURCES

xtext.asm
xtext.inc
xlib.inc
model.inc
xprintf.c

C HEADER FILE

xtext.h

MACROS

FONT_8x8 0
FONT_8x15 1
FONT_USER 2

EXPORTED VARIABLES

NOTE: All variables are read only. If you modify them the results may be unpredictable.

CharHeight BYTE - Height of current inbuilt character set.

CharWidth BYTE - Width of current inbuilt character set.

FirstChar BYTE - First character of current inbuilt character set.

UserCharHeight BYTE - Height of current user character set.

UserCharWidth BYTE - Width of current user character set.

UserFirstCh BYTE - First character of current user character set.

EXPORTED FUNCTIONS

x_text_init

C Prototype: extern WORD x_text_init(void);

Initializes the Mode X text driver and sets the default font (VGA ROM 8x8)

x_set_font

C Prototype: extern void x_set_font(WORD FontId);

FontId - The font number you wish to use (See below)

Select the working font where 0 = VGA ROM 8x8, 1 = VGA ROM 8x14 2 = User defined bitmapped font.

WARNING: A user font must be registered before setting *FontID* to 2.

See Also: Defines for this module.

x_register_userfont

C Prototype: extern void x_register_userfont(char far *UserFontPtr);

UserFontPtr - A pointer to the user font structure.

Register a user font for later selection. Only one user font can be registered at any given time. Registering a user font deregisters the previous user font. User fonts may be at most 8 pixels wide.

USER FONT STRUCTURE

Word: ASCII code of first char in font
Byte: Height of chars in font
Byte: Width of chars in font
n*h*Byte: the font data where n = number of chars and h = height of chars

WARNING: The onus is on the program to ensure that all characters drawn whilst this font is active, are within the range of characters defined.

x_char_put

C Prototype: extern void x_char_put(char ch,WORD X,WORD Y,WORD PgOffs,
WORD Color);

ch - Char to draw

x, y - Screen coordinates at which to draw ch

ScrnOffs - Starting offset of page on which to draw

Color - Color of the text

Draw a text character at the specified location with the specified color.

WARNING: InitTextDriver must be called before using this function

x_printf

C Prototype: void x_printf(int x, int y, unsigned ScrnOffs, int color, char *ln,...);

x, y - screen coordinates at which to draw ch

ScrnOffs - Starting offset of page on which to draw

Color - Color of the text

ln - A pointer to a text string containing formating codes.

Parameters beyond *Color* conform to the standard printf parameters.

Display formated text in the specified color.

x_bgprintf

C Prototype: void x_bgprintf(int x, int y, unsigned ScrnOffs, int fgcolor, int bgcolor, char *ln,...);

x, y - Screen coordinates at which to draw ch.

ScrnOffs - Page offset on which to draw.

fgcolor - Color of the text foreground.

bgcolor - Color of the text background.

ln - Pointer to a text string that contains formating commands that conform to the printf commands.

Parameters beyond *bgcolor* conform to the standard printf parameters.

Display formatted text in the specified foreground and background colors.

x_get_char_width

C Prototype: unsigned int x_get_char_width(char ch)

ch - Character to get the width of.

Returns: The width the requested character.

MODULE XPBITMAP

This module implements a set of functions to operate on planar bitmaps. Planar bitmaps as used by these functions have the following structure:

BYTE 0	The bitmap width in bytes (4 pixel groups) range 1..255
BYTE 1	The bitmap height in rows range 1..255
BYTE 2..n1	The plane 0 pixels width*height bytes
BYTE n1..n2	The plane 1 pixels width*height bytes
BYTE n2..n3	The plane 2 pixels width*height bytes
BYTE n3..n4	The plane 3 pixels width*height bytes

These functions provide the fastest possible bitmap blts from system ram to video and further, the single bitmap is applicable to all pixel alignments. The masked functions do not need separate masks since all non zero pixels are considered to be masking pixels, hence if a pixel is 0 the corresponding screen destination pixel is left unchanged.

SOURCES

xpbitmap.asm
xpbitmap.inc
xlib.inc
model.inc

C HEADER FILE

xpbitmap.h

EXPORTED FUNCTIONS

x_put_masked_pbm

C Prototype: extern void x_put_masked_pbm(WORD X, WORD Y, WORD ScrnOffs,
BYTE far * Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - Pointer to the planar bitmap structure.

Mask write a planar bitmap from system ram to video ram. All zero source bitmap bytes indicate destination byte to be left unchanged.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: No clipping is supported. Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

x_put_pbm

C Prototype: extern void x_put_pbm(WORD X, WORD Y, WORD ScrnOffs,
BYTE far *Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - Pointer to the planar bitmap structure.

Write a planar bitmap from system ram to video ram.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: No clipping is supported Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

x_get_pbm

C Prototype: extern void x_get_pbm(WORD X, WORD Y,BYTE Bw,BYTE Bh,
WORD ScrnOffs, BYTE far * Bitmap);

X, Y - Coordinates of the upper left corner of the bitmap.

Bw - Width of the bitmap to get.

Bh - Height of the bitmap to get.

ScrnOffs - Page offset to get the bitmap from.

Bitmap - Pointer to a buffer allocated for receiveing this bitmap.

Read a planar bitmap to system ram from video ram.

NOTE: Width is in bytes in lots of 4 pixels

LIMITATIONS: No clipping is supported. Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

MODULE XPBMCLIP

This module implements a similar set of functions to operate on planar bitmaps as XPBITMAP but incorporates clipping to a user defined clipping rectangle (which is set by ***x_set_cliprect*** in module XMAIN).

The planar bitmap format is identical to the above module

There are three variations of the functions in XPBITMAP in this module identified by the three function name extensions: _clipx, _clipy _clipxy. Because speed is critical in games programming you do not want to be checking for clipping if not necessary thus for sprites that move only horizontally you would use the _clipx version of the put function, for sprites that move vertically you would use the _clipy version and for sprites that move both directions you would use the clipxy version. Keep in mind also that the clipping components of these functions assume that the clipping rectangle is equal to or larger than the size of the bitmap i.e.. if a bitmap is top clipped, it is assumed that the bitmap's bottom is not also clipped. Similarly with horizontal clipping.

Note: Performance in decreasing order is as follows. _clipy,_clipx,_clipxy with masked puts being slower than unmasked puts.

Horizontal clipping is performed to byte boundaries (4 pixels) rather than pixels. This allows for the fastest implementation of the functions. It is not such a handicap because for one, your screen width a multiple of 4 pixels wide and for most purposes it is the screen edges that form the clipping rectangle.

Following is an example of setting a clipping rectangle to the logical screen edges:

```
x_set_cliprect(0,0,ScrnLogicalByteWidth,ScrnLogicalHeight)
```

NOTE: The functions now return a value; 1 if clipped image is fully clipped (i.e. no portion of it appears on the screen) otherwise it returns 0

SOURCES

xpbmclip.asm
xpbmclip.inc
xlib.inc
model.inc

C HEADER FILE

xpbmclip.h

EXPORTED FUNCTIONS

x_put_masked_pbm_clipx

C Prototype: extern void x_put_masked_pbm_clipx(WORD X, WORD Y, WORD ScrnOffs,
BYTE far * Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - A far pointer to the planar bitmap structure.

Mask write a planar bitmap from system ram to video ram. Horizontal clipping is performed. All zero source bitmap bytes indicate destination byte to be left unchanged.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

x_put_masked_pbm_clipy

C Prototype: extern void x_put_masked_pbm_clipy(WORD X, WORD Y, WORD ScrnOffs,
BYTE far * Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - A far pointer to the planar bitmap structure.

Mask write a planar bitmap from system ram to video ram. Vertical clipping is performed. All zero source bitmap bytes indicate destination byte to be left unchanged.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

x_put_masked_pbm_clipxy

C Prototype: extern void x_put_masked_pbm_clipxy(WORD X, WORD Y, WORD ScrnOffs,
BYTE far * Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - A far pointer to the planar bitmap structure.

Mask write a planar bitmap from system ram to video ram. Both horizontal and vertical clipping is performed. All zero source bitmap bytes indicate destination byte to be left unchanged.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

x_put_pbm_clipx

C Prototype: extern void x_put_pbm_clipx(WORD X, WORD Y, WORD ScrnOffs,
BYTE far *Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - A far pointer to the planar bitmap structure.

Write a planar bitmap from system ram to video ram. Horizontal clipping is performed.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

x_put_pbm_clipy

C Prototype: extern void x_put_pbm_clipy(WORD X, WORD Y, WORD ScrnOffs,
BYTE far *Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - A far pointer to the planar bitmap structure.

Write a planar bitmap from system ram to video ram. Vertical clipping is performed.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

x_put_pbm_clipxy

C Prototype: extern void x_put_pbm_clipxy(WORD X, WORD Y, WORD ScrnOffs,
BYTE far *Bitmap);

x, y - Coordinates for the upper left corner of the bitmap.

ScrnOffs - Page offset to place the bitmap at.

Bitmap - A far pointer to the planar bitmap structure.

Write a planar bitmap from system ram to video ram. Both horizontal and vertical clipping is performed.

NOTE: Width is in bytes i.e. lots of 4 pixels

LIMITATIONS: Only supports bitmaps with widths which are a multiple of 4 pixels

See Also: XBMTOOLS module for linear <-> planar bitmap conversion functions.

MODULE XCBITMAP

Compiled bitmap functions. See "The care and feeding of compiled masked blits." in the reference section for more details on compiled bitmaps.

SOURCES

xcbitmap.asm
xcbitmap.inc

C HEADER FILE

xcbitmap.h

EXPORTED FUNCTIONS

x_compile_bitmap

C Prototype: int x_compile_bitmap(WORD lsw, char far *bitmap, char far *output);

lsw - The logical screen width in bytes.

bitmap - A pointer to the source linear bitmap.

output - A pointer to a buffer set up to receive the compiled bitmap.

Returns: The size of the compiled bitmap in bytes.

Compiles a linear bitmap to generate machine code to plot it at any required screen coordinates FAST!

x_sizeof_cbitmap

C Prototype: int x_sizeof_cbitmap(WORD lsw, char far *bitmap);

lsw - The logical screen width in bytes.

bitmap - A pointer to the source linear bitmap.

Returns: The space in bytes required to hold the compiled bitmap.

x_put_cbitmap

C Prototype: void x_put_cbitmap(int X, int Y, unsigned int PageOffset, char far *bitmap);

X, Y - The coordinates at which to place the compiled bitmaps.

PageOffset - The page offset on which to draw the bitmap.

bitmap - A pointer to the compiled bitmap.

Displays a bitmap generated by ***x_compile_bitmap***.

MODULE XCOMPPBM

Identical to XCBITMAP except that the source bitmaps are the PBM form rather than LBM.

SOURCES

xcompbm.asm
xcompbm.inc

C HEADER FILE

xcompbm.h

EXPORTED FUNCTIONS

x_compile_pbm

C Prototype: x_compile_pbm(WORD lsw, char far *bitmap, char far *output);

lsw - The logical screen width in bytes.

bitmap - A pointer to the source planar bitmap.

output - A far pointer to a buffer set up to receive the compiled bitmap.

Returns: The size of the compiled bitmap in bytes.

Compiles a planar bitmap to generate machine code to plot it at any required screen coordinates FAST!

x_sizeof_cpbm

C Prototype: int x_sizeof_cpbm(WORD lsw, char far *bitmap);

lsw - The logical screen width in bytes.

bitmap - A far pointer to the source planar bitmap.

Returns: The space in bytes required to hold the compiled bitmap.

MODULE XVBITMAP

The XVBITMAP module implements yet another type of bitmap to complement planar and compiled bitmaps, VRAM based bitmaps. If a 4 cylinder car is analogous to planar bitmaps, that is thrifty on memory consumption but low performance and a V8 is analogous to Compiled bitmaps, memory guzzlers that really fly, then VRAM based bitmaps are the 6 cylinder modest performers with acceptable memory consumption.

To summarize their selling points, VBM's are moderately fast with fair memory consumption, and unlike compiled bitmaps, can be clipped. The disadvantages are that they are limited by the amount of free video ram and have a complex structure.

The VRAM bitmap format is rather complex consisting of components stored in video ram and components in system ram working together. This complexity necessitates the existence of a creation function ***x_make_vbm*** which takes an input linear bitmap and generates the equivalent VBM (VRAM Bit Map).

VBM structure:

```
        WORD   0    Size              Total size of this VBM structure in bytes
        WORD   1    ImageWidth        Width in bytes of the image (for all alignments)
        WORD   2    ImageHeight       Height in scan lines of the image
        WORD   3    Alignment 0   ImagePtr    Offset in VidRAM of this aligned image
        WORD   4                  MaskPtr     Offset (within this structure's DS) of alignment
masks
        WORD   9    Alignment 3   ImagePtr    Offset in VidRAM of this aligned image
        WORD 10                   MaskPtr     Offset (within this structure's DS) of alignment
masks
        BYTE    21 (WORD 11)
                                      Image masks for alignment 0
        BYTE   21 + ImageWidth*ImageHeight
                                      (similarly for alignments 1 - 2 )
        BYTE   21 + 3*ImageWidth*ImageHeight + 1
                                      Image masks for alignment 3
        BYTE   21 + 4*(ImageWidth*ImageHeight)
                                      Similarly for alignments 2 and 3
        BYTE 21 + 4*(ImageWidth*ImageHeight)
                                      (And don't forget the corresponding data in video ram)
```

You can see for yourself the complexity of this bitmap format. The image is stored in video ram in its 4 different alignments with pointers to these alignments in the VBM. Similarly there are 4 alignments of the corresponding masks within the VBM itself (towards the end). The mask bytes contain the plane settings for the corresponding video bytes so that one memory move can move up to 4 pixels at a time (depending on the mask settings) using the VGA's latches, theoretically giving you a 4x speed improvement over conventional blits like the ones implemented in

"XPBITMAP". In actual fact its anywhere between 2 and 3 due to incurred overheads.

These bitmaps are more difficult to store in files than PBM'S and CBM's but still possible with a bit of work, so do not dismiss these as too difficult to use. Consider all the bitmap formats carefully before deciding on which to use. There may even be situations that a careful application of all three types would be most effective i.e.. compiled bitmaps for Background tiles and the main game character (which never need clipping), VRAM based bitmaps for the most frequently occurring (opponent, alien etc.) characters which get clipped as they come into and leave your current location and planar bitmaps for smaller or less frequently encountered characters.

SOURCES

xvbitmap.asm
xvbitmap.inc
xlib.inc model.inc
xmakevbm.c

C HEADER FILE

xvbitmap.h

EXPORTED FUNCTIONS

x_make_vbm

C Prototype: extern char far * x_make_vbm(char far *lbm, WORD *VramStart);

lbm - A far pointer to the input linear bitmap

VramStart - Pointer to variable containing Offset of first free VRAM byte

Create the VBM from the given linear bitmap and place the image alignments in video ram starting at the offset in the variable pointed to by *VramStart*. *VramStart* is then updated to point to the next free VRAM byte (just after the last byte of the image alignments). Usually you will point *VramStart* to *NonVisual_Offs*.

x_put_masked_vbm

C Prototype: extern int x_put_masked_vbm(int X, int Y, WORD ScrnOffs,
BYTE far *VBitmap);

X, Y - Coordinates to draw the bitmap at.

ScrnOffs - The page offset to draw the bitmap at.

VBitmap - A far pointer to the video bitmap.

Returns: 1 if clipped image is fully clipped (i.e. no portion of it appears on the screen) otherwise it returns 0. (Editors note: since this function doesn't support clipping the return value should be 0 at all times.)

Draw a VRAM based bitmap at (X,Y) relative to the screen with starting offset *ScrnOffs*.

x_put_masked_vbm_clipx

C Prototype: extern int x_put_masked_vbm_clipx(int X, int Y, WORD ScrnOffs,
BYTE far *VBitmap);

X, Y - Coordinates to draw the bitmap at.

ScrnOffs - The page offset to draw the bitmap at.

VBitmap - A far pointer to the video bitmap.

Returns: 1 if clipped image is fully clipped otherwise it returns 0.

Draw a VRAM based bitmap at (X,Y) relative to the screen with starting offset *ScrnOffs*. Horizontal clipping is performed.

x_put_masked_vbm_clipy

C Prototype: extern int x_put_masked_vbm_clipy(int X, int Y, WORD ScrnOffs,
BYTE far *VBitmap);

X, Y - Coordinates to draw the bitmap at.

ScrnOffs - The page offset to draw the bitmap at.

VBitmap - A far pointer to the video bitmap.

Returns: 1 if clipped image is fully clipped (i.e. no portion of it appears on the screen) otherwise it returns 0

Draw a VRAM based bitmap at (X,Y) relative to the screen with starting offset *ScrnOffs*. Vertical clipping is performed.

x_put_masked_vbm_clipxy

C Prototype: extern int x_put_masked_vbm_clipxy(int X, int Y, WORD ScrnOffs,
BYTE far *VBitmap);

X, Y - Coordinates to draw the bitmap at.

ScrnOffs - The page offset to draw the bitmap at.

VBitmap - A far pointer to the video bitmap.

Returns: 1 if clipped image is fully clipped (i.e. no portion of it appears on the screen) otherwise it returns 0

Draw a VRAM based bitmap at (X,Y) relative to the screen with starting offset *ScrnOffs*.
Both horizontal and vertical clipping is performed.

See XPBMCLIP for more details on the type of clipping used as it is identical to XVBITMAP.

MODULE XMOUSE

The XMOUSE module implements very basic mouse handling functions. The way in which it operates is by installing an event handler function during initialization which subsequently intercepts and processes mouse events and automatically updates status variables such as mouse position and button pressed status.

It does not support the full functionality of:
SPLIT SCREENS, SCROLLED WINDOWS, or VIRTUAL WINDOWS
This was done to primarily prevent unnecessary impedences to performance, since the mouse handler function has the potential to degrade performance. It also saves me a lot of coding which I was too lazy to do.

Programs communicate with the mouse driver as with other devices, through an interrupt vector namely 33h. On generating an interrupt, the mouse driver expects a function number in AX and possibly other parameters in other registers and returns information via the registers. A brief description of the mouse functions follows:

MS Mouse Driver Functions

Mouse Initialization	0
Show Cursor	1
Hide Cursor	2
Get Mouse Position & Button Status	3
Set Mouse Cursor Position	4
Get Button Press Information	5
Get Button Release Information	6
Set Min/Max Horizontal Position	7
Set Min/Max Vertical Position	8
Define Graphics Cursor Block	9
Define Text Cursor	10
Read Mouse Motion Counters	11
Define Event Handler	12
Light Pen Emulation Mode ON	13
Light Pen Emulation Mode OFF	14
Set Mouse Mickey/Pixel Ratio	15
Conditional Hide Cursor	16
Set Double-Speed Threshold	19

In practice only a few of these functions are used and even fewer when the mouse status is monitored by an event handler function such as is used in this module.

The most important thing to note when using the mouse module is that the mouse event handler must be removed before exiting the program. It is a good idea to have an exit function (see the C ***atexit*** function) and include the line ***x_mouse_remove***(); along with any other pre-exit cleanup code.

SOURCES

xmouse.asm
xlib.inc
model.inc

C HEADER FILE

xmouse.h

EXPORTED VARIABLES

MouseInstalled WORD - Indicates whether mouse handler installed

MouseHidden WORD - Indicates whether mouse cursor is hidden

MouseButtonStatus WORD - Holds the mouse button status

MouseX WORD - Current X position of mouse cursor

MouseY WORD - Current Y position of mouse cursor

MouseFrozen WORD - Disallows position updates if TRUE

MouseColor BYTE - The mouse cursors color

EXPORTED FUNCTIONS

x_mouse_init

C Prototype: int x_mouse_init()

Initialize the mouse driver functions and install the mouse event handler function. This is the first function you must call before using any of the mouse functions. This mouse code uses the fastest possible techniques to save and restore mouse backgrounds and to draw the mouse cursor.

WARNING: This function uses and updates *NonVisual_Offset* to allocate video ram for the saved mouse background.

LIMITATIONS: No clipping is supported horizontally for the mouse cursor. No validity checking is performed for *NonVisual_Offs*

****WARNING**** You must Hide or at least Freeze the mouse cursor while drawing using any of the other XLIB modules since the mouse handler may modify VGA register settings at any time. VGA register settings are not preserved which will result in unpredictable drawing behavior. If you know the drawing will occur away from the mouse cursor set *MouseFrozen* to TRUE (1), do your drawing then set it to FALSE (0). Alternatively call ***x_hide_mouse***, perform your drawing and then call ***x_show_mouse***. Another alternative is to disable interrupts while drawing but usually drawing takes up a lot of time and having interrupts disabled for too long is not a good
idea.

x_define_mouse_cursor

C Prototype: void x_define_mouse_cursor(char far *MouseDef, unsigned char MouseColor)

MouseDef - A far pointer to 14 characters containing a bitmask for all the cursor's rows.

MouseColor - The color to use when drawing the mouse cursor.

Define a mouse cursor shape for use in subsequent cursor redraws. XMOUSE has a hardwired mouse cursor size of 8 pixels across by 14 pixels down.

WARNING: This function assumes *MouseDef* points to 14 bytes.

Note: Bit order is in reverse. i.e. bit 7 represents pixel 0, bit 0 represents pixel 7 in each byte of *MouseDef* .

x_show_mouse
C Prototype: void x_show_mouse();

Makes the cursor visible if it was previously hidden.

See Also: ***x_hide_mouse***.

x_hide_mouse
C Prototype: void x_hide_mouse();

Makes the cursor hidden if it was previously visible.

See Also: ***x_show_mouse***.

x_mouse_remove
C Prototype: void x_mouse_remove();

Stop mouse event handling and remove the mouse handler.

NOTE: This function MUST be called before quitting the program if a mouse handler has been installed

x_position_mouse
C Prototype: void x_position_mouse(int x, int y);

x, y - Coordinates to move the mouse cursor to.

Positions the mouse cursor at the specified location

x_mouse_window
C Prototype: void x_mouse_window(int x0, int y0, int x1, int y1);

x0, y0 - Coordinates of the upper left corner of the window.

x1, y1 - Coordinates of the lower right corner of the window.

Defines a mouse window. The mouse cursor is unable to move from this window.

x_update_mouse

C Prototype: void x_update_mouse();

Forces the mouse position to be updated and cursor to be redrawn.

Note: This function is useful when you have set *MouseFrozen* to true.
Allows the cursor position to be updated manually rather than automatically by the installed handler.

MODULE XBMTOOLS

This module implements a set of functions to convert between planar bitmaps and linear bitmaps.

PLANAR BITMAPS

Planar bitmaps as used by these functions have the following structure:

BYTE 0	The bitmap width in bytes (4 pixel groups) range 1..255
BYTE 1	The bitmap height in rows range 1..255
BYTE 2..n1	The plane 0 pixels width*height bytes
BYTE n1..n2	The plane 1 pixels width*height bytes
BYTE n2..n3	The plane 2 pixels width*height bytes
BYTE n3..n4	The plane 3 pixels width*height bytes

as used by ***x_put_pbm, x_get_pbm, x_put_masked_pbm***.

LINEAR BITMAPS

Linear bitmaps have the following structure:

BYTE 0	The bitmap width in pixels range 1..255
BYTE 1	The bitmap height in rows range 1..255
BYTE 2..n	The width*height bytes of the bitmap

SOURCES

xbmtools.asm
xpbmtools.inc
model.inc

C HEADER FILE

xbmtools.h

MACROS

BM_WIDTH_ERROR

LBMHeight(lbitmap) - Height of linear bitmap "lbitmap"

LBMWidth(lbitmap) - Width of linear bitmap "lbitmap"

PBMHeight(pbitmap) - Height of planar bitmap "pbitmap"

PBMWidth(pbitmap) - Width of planar bitmap "pbitmap"

LBMPutPix(x, y, lbitmap, color) - Set the color of pixel (x, y) color in linear bitmap

LBMGetPix(x, y, lbitmap) - Get the color of pixel (x, y) in linear bitmap

EXPORT FUNCTIONS

x_pbm_to_bm

C Prototype: extern int x_pbm_to_bm(char far * source_pbm, char far * dest_bm);

source_pbm - A pointer to the source planar bitmap.

dest_bm - A pointer to a buffer set up to receive the linear bitmap.

Returns: 0 on successful conversion.

This function converts a bitmap in the planar format to the linear format as used by ***x_compile_bitmap***.

WARNING: The source and destination bitmaps must be pre-allocated.

NOTE: This function can only convert planar bitmaps that are suitable. If the source planar bitmap's width (per plane) is >= 256/4 it cannot be converted. In this situation an error code BM_WIDTH_ERROR.

x_bm_to_pbm

C Prototype: extern int x_bm_to_pbm(char far * source_lbm, char far * dest_bm);

source_lbm - A pointer to the source linear bitmap.

dest_bm - A pointer to a buffer set up to receive the planar bitmap.

Returns: 0 on successful conversion.

This function converts a bitmap in the linear format as used by x_compile_bitmap to the planar format.

WARNING: The source and destination bitmaps must be pre - allocated

NOTE: This function can only convert linear bitmaps that are suitable. If the source linear bitmap's width is not a multiple of 4 it cannot be converted. In this situation an error code BM_WIDTH_ERROR.

MODULE XCLIPPBM

Note: VERY SIMILAR to XPBMCLIP. This module implements blits of clipped planar bitmaps. Blits are clipped to pixels, both horizontally. This makes the unmasked blit function here slightly slower than the equivalent functions in the XPBMCLIP module.

SOURCES

xclippbm.asm
xclippbm.inc
xlib.inc
model.inc

C HEADER FILE

xclippbm.h

EXPORTED VARIABLES

TopBound - int

BottomBound - int

LeftBound - int

RightBound - int

EXPORTED FUNCTIONS

x_clip_pbm

C Prototype: extern int x_clip_pbm (int X, int Y, int ScreenOffs, char far * Bitmap);

X, Y - The coordinates to place the bitmap at.

ScreenOffset - The offset of the page on which to draw the bitmap.

Bitmap - A far pointer to the planar bitmap.

Returns: If the entire bitmap turns out to be outside the bounding box, this function returns a 1, otherwise it returns a 0

Copies a planar bitmap from SRAM to VRAM, with clipping.

.

x_clip_masked_pbm

C Prototype: extern int x_clip_masked_pbm (int X, int Y, int ScreenOffs, char far * Bitmap);

X, Y - The coordinates to place the bitmap at.

ScreenOffset - The offset of the page on which to draw the bitmap.

Bitmap - A far pointer to the planar bitmap.

Returns: If the entire bitmap turns out to be outside the bounding box, this function returns a 1, otherwise it returns a 0

Copies a planar bitmap from SRAM to VRAM, with clipping, 0 bytes in the bitmap are not copied.

MODULE XCIRCLE

Mode X circle functions

SOURCES

xcircle.asm
xcircle.inc
xlib.inc
model.inc

C HEADER FILE

xcircle.h

EXPORTED FUNCTIONS

x_circle

C Prototype: extern void x_circle (WORD Left, WORD Top, WORD Diameter,
WORD Color, WORD ScreenOffs);

Left, Top - The coordinates of the upper left corner of the circle, in pixels.

Diameter - The diameter of the circle.

Color - The color in which to draw the circle.

ScreenOffs - The page offset to draw the circle on.

Draws a circle with the given upper-left-hand corner and diameter, which are given in pixels.

x_filled_circle

C Prototype: extern void x_filled_circle (WORD Left, WORD Top, WORD Diameter,
WORD Color, WORD ScreenOffs);

Left, Top - The coordinates of the upper left corner of the circle, in pixels.

Diameter - The diameter of the circle.

Color - The color in which to draw the circle.

ScreenOffs - The page offset to draw the circle on.

Draws a filled circle with the given upper-left-hand corner and diameter.

MODULE XDETECT

This module implements a set of functions to detect the PC's hardware configuration.

SOURCES

xdetect.asm
xdetect.inc
model.inc

C HEADER FILE

xdetect.h

EXPORTED MACROS

I8086 0
I80186 1
I80286 2
I80386 3

NoGraphics 0
MDA 1
CGA 2
EGAMono 3
EGAColor 4
VGAMono 5
VGAColor 6
MCGAMono 7
MCGAColor 8

BUS_MOUSE 1
SERIAL_MOUSE 2
INPORT_MOUSE 3
PS2_MOUSE 4
HP_MOUSE 5

EXPORTED VARIABLES

MouseButtonCount WORD - The number of buttons on the detected mouse

MouseVersion WORD - Mouse driver version
(High byte = Major version, Low byte = minor version)

MouseType BYTE - The mouse type

MouseIRQ BYTE - The IRQ number used by the mouse driver

EXPORTED FUNCTIONS

x_graphics_card

C Prototype: extern int x_graphics_card();

Returns: The type of graphics card installed.

See the defines for this module.

x_processor

C Prototype: extern int x_processor();

Returns: The type of processor installed.

Note: A 486 registers as a 386.

See defines for this module.

x_coprocessor

C Prototype: extern int x_coprocessor();

Returns: 1 of a numeric co-processor is present, 0 if not.

Note: The type is not detected but it's not really necessary as the processor type usually determines the numeric coprocessor type

x_mousedriver

C Prototype: extern int x_mousedriver();

Returns: 1 of a mouse driver is installed, 0 otherwise.

If a mouse driver is detected the mouse related variables are set accordingly.

MODULE XFILEIO

Handle based file I/O functions.
See any good DOS programming reference for more information on int 21h DOS services.

SOURCES

xfileio.asm
xfileio.inc
model.inc

C HEADER FILE

xfileio.h

EXPORTED MACROS

File access modes

F_RDONLY - Read only.
F_WRONLY - Write only.
F_RDWR - Read and write

Seek codes

SEEK_START - Bytes from the start of the file.
SEEK_CURR - Bytes from the current position.
SEEK_END - Bytes from the end of the file.

File error value

FILE_ERR

EXPORTED FUNCTIONS

f_open

C Prototype: extern int f_open(char * filename, char access);

filename - A pointer to a string containing the full path and filename you wish to open.

access - A character that defines the access mode.

Returns: The file handle on success, FILE_ERR on failure.

Opens a file according to the access char.

f_close

C Prototype: extern int f_close(int handle);

handle - The handle of the file you wish to close.

Returns: 0 on success, FILE_ERR on failure.

Closes the file associated with the specified handle.

f_read

C Prototype: extern int f_read(int handle,char near * buffer, int count);

handle - The handle of the file you wish to read from.

buffer - A near pointer to a buffer to receive the bytes from the file.

count - The number of bytes to read from the file.

Returns: The number of bytes read on success, FILE_ERR on failure

Reads bytes from the a file into a near buffer

f_readfar

C Prototype: extern int f_readfar(int handle,char far * buffer, int count);

handle - The handle of the file you wish to read from.

buffer - A far pointer to a buffer set up to receive the data.

count - The number of bytes to read.

Returns: The number of bytes read on success, FILE_ERR on failure

Reads a block of bytes a file into a far buffer.

f_write

C Prototype: extern int f_write(int handle, char near * buffer, int count);

handle - The handle of the file you wish to write to.

buffer - A near pointer to a buffer containing the data to write.

count - The number of bytes to write.

Returns: The number of bytes written on success, FILE_ERR on failure

Writes a block bytes to a file from a near buffer

f_writefar

C Prototype: extern int f_write(int handle, char far * buffer, int count);

handle - The handle of the file you wish to write to.

buffer - A far pointer to a buffer containing the data to write.

count - The number of bytes to write.

Returns: The number of bytes written on success, FILE_ERR on failure

Writes a block of bytes to a file from a far buffer.

f_seek

C Prototype: extern long int f_seek(int handle, long int position, char method_code)

handle - The handle of the file you are working with.

position - The position to move to.

method_code - The seek method you wish to use. (See the defines.)

Returns: The file pointer position on success, FILE_ERR on failure.

Moves the file pointer.

f_filelength

C Prototype: extern long int f_filelength(int handle)

handle - The handle of the file you are working with.

Returns: The length of the file on success, FILE_ERR on failure.

f_tell

C Prototype: extern long int f_tell(int handle)

handle - The handle of the file you are working with.

Returns: The file pointer position on success, FILE_ERR on failure

MODULE XRLETOOL

This module implements a number of functions comprising an RLE encoding decoding system.

SOURCES

xrletool.asm
xrletool.inc
model.inc

C HEADER FILE

xrletool.h

EXPORTED FUNCTIONS

x_buff_RLDecode

C prototype: extern unsigned int x_buff_RLDecode(char far * source_buff, char far *dest_buff);

source_buff - A far pointer to the buffer to compress

dest_buff - A far pointer to the destination buffer

Returns: The size of the resultant uncompressed data.

Expands an RLE compressed source buffer to a destination buffer.

WARNING: The buffers must be pre-allocated.

x_buff_RLEncode

C prototype: extern unsigned int x_buff_RLEncode(char far * source_buff, char far *dest_buff,
unsigned int count);

source_buff - A far pointer to the buffer to compress.

dest_buff - A far pointer to the destination buffer.

count - The size of the source data in bytes.

Returns: The size of the resultant compressed data.

RLE Compresses a source buffer to a destination buffer.

WARNING: The buffers must be pre allocated.

x_buff_RLE_size

C prototype: extern unsigned int x_buff_RLE_size(char far * source_buff, unsigned int count);

source_buff - A far pointer to the uncompressed data buffer

count - The size of the source data in bytes

Returns: The size the input data would compress to.

x_file_RLEncode

C prototype: extern unsigned int x_file_RLEncode(int handle, char far * source_buff,
unsigned int count);

source_buff - A far pointer to the buffer to compress

handle - The output file handle.

count - The size of the source data in bytes

Returns: The size of the resultant compressed data or 0 if it fails.

RLE Compresses a source buffer to an output file.

x_file_RLDecode

C prototype: extern unsigned int x_buff_RLDecode(int handle, char far * dest_buff);

handle - Input file handle

dest_buff - A far pointer to the destination buffer

Returns: The size of the resultant uncompressed data.

Expands an RLE compresses file to a destination RAM buffer.

MODULE XPOLYGON

This module implements general filled convex polygon and triangle functions

SOURCES

xpolygon.asm
xpolygon.inc

C HEADER FILE

xpolygon.h

TYPE DEFS

typedef struct { int X; int Y; } far VERTEX;

EXPORTED FUNCTIONS

x_triangle

C Prototype: void x_triangle(int x0, int y0, int x1, int y1, int x2, int y2, WORD color, WORD PageBase);

x0, y0 - The coordinates of point one of the triangle.

x1, y1 - The coordinates of point two of the triangle.

x2, y2 - The coordinates of the third (and final) point of the triangle.

color - The color in which to draw the triangle in.

PageBase - The page offset on which to draw the triangle.

This function draws a filled triangle which is clipped to the current clipping window defined by *TopClip, BottomClip, LeftClip, RightClip*.

Remember: The X clipping variable are in BYTES not PIXELS so you can only clip to 4 pixel byte boundaries.

x_polygon

C Prototype: void x_polygon(VERTEX *vertices, int num_vertices, WORD color,
WORD PageBase);

vertices - A pointer to your vertex structure.

num_vertices - The number of vertices to plot.

color - The color in which to draw the polygon.

PageBase - The page offset on which to draw the polygon.

This function is similar to the triangle function but draws convex polygons. The vertices are supplied in the form of a FAR pointer.

NOTE: A convex polygon is one such that if you draw a line from any two vertices, every point on that line will be within the polygon.

This function works by splitting up a polygon into its component triangles and calling the triangle routine above to draw each one. Performance is respectable but a custom polygon routine might be faster.

MODULE XBEZIER

Bezier curve plotting function. See "What is a Bezier curve?" for more details.

SOURCES

xbezier.asm
xlib.inc

C HEADER FILE

xbezier.h

EXPORTED FUNCTIONS

x_bezier

C Prototype: void x_bezier(int e1x, int e1y, int c1x, int c1y, int c2x, int c2y,
int e2x, int e2y, int levels, char color, WORD PageBase);

e1x, e1y, e2x, e2y - The coordinates of the two endpoints.

c1x, c1y, c2x, c2y - The coordinates of the two control points.

levels -

color - The color to draw with.

PageBase - The page offset on which to draw.

Plots a Bezier curve on the screen.

See also: "What is a Bezier curve?"

MODULE XFILL

This module implements a couple of general purpose flood fill functions.

SOURCES

xfill.asm
xlib.inc

C HEADER FILE

xfill.h

EXPORTED FUNCTIONS

x_flood_fill

C Prototype: int x_flood_fill(int x, int y, WORD offs, int boundary, int color);

x, y - The starting coordinates.

offs - The page offset to draw on.

boundary - *Boundary color.*

color - The color to use for the fill.

Returns: The number of pixels that were filled.

This function performs the familiar flood filling used by many paint programs and, of course, the Borland BGI's flood fill function. The pixel at *x, y* and all adjacent pixels of the same color are filled with the new color. Filling stops when there are no more adjacent pixels of the original pixels color.

x_boundry_fill

C Prototype: int x_boundry_fill(int x, int y, WORD offs, int boundary, int color);

x, y - The starting coordinates.

offs - The page offset to draw on.

boundary - *Boundary color.*

color - The color to use for the fill.

Returns: The number of pixels that were filled.

This function performs a variant of the flood fill described above. The pixel at *x, y* and all adjacent pixels of the same color are filled with the new color. Filling stops when the area being filled is fully enclosed by pixels of the color *boundary.*

MODULE XVSYNC

Inspired by REND386 v3.01 by Dave Stamps and Bernie Roehl

This module uses timer 0 to simulate a vertical retrace interrupt. It's designed to significantly reduce the idle waiting time in XLIB. Why simulate the VRT interrupt? Simply because a true VRT interrupt is not implemented on many VGA cards. Using a VRT interrupt as opposed to polling, can result in huge performance improvements for your code and help make animation much smoother than it would be using polling.

Normally XLIB waits for vsync when ***x_page_flip, x_set_start_address*** or ***x_put_pal_???*** is called. the waiting period could be better utilized to do housekeeping calculations or whatever. The ***x_put_pal_???*** functions also don't work very smoothly in conjunction with other functions that wait for the vertical retrace since each function introduces its own VRT delay.

When using the vsync handler, the VRT delay is reduced to the absolute minimum which can result in a huge performance boost for your programs.

When using double buffering, you may still have to wait before drawing, but you could do as much other work as possible, like this:

```
x_page_flip(...);
/* animate the palette */
/* do some collision detection and 3D calculations */
/* read the joystick */
x_wait_start_address();     /* Not needed with triple buffering. */
/* draw the next frame. */
```

SOURCES

xvsync.asm
xmain.asm
xvsync.inc
xmain.inc

C HEADER FILE

xvsync.h

EXPORTED VARIABLES

VsyncPeriod WORD - Time in 1.193 between two vsyncs.

TicksPerSecond WORD - Number of vsyncs per second.

VsyncTimerInt long - Number of vsyncs since ***x_install_vsync_handler*** was called. Nice for game timing.

EXPORTED FUNCTIONS

x_install_vsync_handler

C Prototype: void x_install_vsync_handler(int VRTsToSkip);

VRTsToSkip - Defines the delay in VRT's between consecutive physical start address changes, thus allowing you to limit the maximum frame rate for page flips in animation systems. The frame rate is calculated as Vertical Refresh Rate / *VRTsToSkip*. Thus in the 320x240 mode which refreshes at 60Hz a *VRTsToSkip* value of 3 will result in a maximum page flipping rate of 20Hz (Frames per second).

This function installs the vsync handler using timer 0. It's called about 100 microseconds before every vertical retrace.

WARNING: Be sure to remove it before exiting your program.
When used with a debugger, the system clock may speed up.

x_remove_vsync_handler

C Prototype: void x_remove_vsync_handler(void);

This function removes the vsync handler.

You must call this function before exiting your program, or your system will crash!

x_set_user_vsync_handler

C Prototype: void x_set_user_vsync_handler(void far (*f)());

f - A far pointer to a user function to be called once each vertical retrace.

This function installs a user routine to be called once each vertical retrace. The user handler has its own stack of 256 bytes, so be careful with the stack checking option in BC.

WARNING: This installs an interrupt driven handler, beware of the following:

- ◆Only 8086 registers are preserved. If you're using 386 code, save all the 386 registers.
- ◆Don't do any drawing.
- ◆Don't call any DOS functions.

You CAN update global variables if you're careful. And it's nice for palette animation. You can even do fades while loading from disk. You should usee this instead of installing your own int 08h routine and chaining to the original.

x_wait_start_addr

C Prototype: void x_wait_start_addr(void);

You must call this function before drawing after a call to ***x_set_start_addr*** or ***x_page_flip*** when you are using the vsync handler and not using triple buffering.

MODULE XCBITM32

This module implements 32 bit compiling of linear bitmaps. There are no functions for plotting the compiled bitmaps in this module, use ***x_put_cbitmap*** for that.

SOURCES

xcbitm32.c

C HEADER FILE

xcbitm32.h

EXPORTED FUNCTIONS

x_compile_bitmap_32

C Prototype: x_compile_bitmap_32(WORD lsw, char far *bitmap, char far *output);

lsw - The logical screen width in bytes.

bitmap - A pointer to the source linear bitmap.

output - A far pointer to a buffer set up to receive the compiled bitmap.

Returns: The size of the compiled bitmap in bytes.

Compiles a linear bitmap to generate 386+ machine code to plot it at any required screen coordinates FAST!

x_sizeof_cbitmap_32

C Prototype: int x_sizeof_cbitmap32(WORD lsw, char far *bitmap);

lsw - The logical screen width in bytes.

bitmap - A far pointer to the source linear bitmap.

Returns: The space in bytes required to hold the compiled bitmap.

REFERENCE SECTION

REFERENCES

In my opinion Doctor Dobbs Journal is the best reference text for VGA Mode X graphics:

Issue 178 Jul. 1991 : First reference to Mode X Article Abstract : VGA's undocumented Mode X supports page flipping, makes off screen memory available, has square pixels, and increases performance by as much as 4 times.

Issue 179 Aug. 1991 : Continuation Article Abstract. Michael discusses latches and VGA's undocumented Mode X.

Issue 181 Sept. 1991 : Continuation Article Abstract. Michael puts the moves on animation using VGA's 256 colors.

Issue 184 Oct. 1991 : First of a continuing series covering 3-D animation using VGA's Mode X. This series is still ongoing

(October 1992) Article Abstract. Michael moves into 3-D animation, starting with basic polygon fills and page flips.

WHAT IS MODE X ?

Mode X is a derivative of the VGA's standard mode 13h (320x200 256 color). It is a (family) of undocumented video modes that are created by tweaking the VGA's registers. The beauty of mode X is that it offers several benefits to the programmer: - Multiple graphic pages where mode 13h doesn't allowing for page flipping (also known as double buffering) and storage of images and data in offscreen video memory - A planar video ram organization which although more difficult to program, allows the VGA's plane-oriented hardware to be used to process pixels in parallel, improving performance by up to 4 times over mode 13h

WHAT IS A SPLIT SCREEN ?

A split screen is a hardware feature offered by the EGA and VGA video cards. A split screen is a mode of graphics operation in which the hardware splits the visual graphics screen horizontally and treats both halves as individual screens each starting at different locations in video RAM. The bottom half (which is usually referred to as the split screen) always starts at address A000:0000 but the top half's starting address is user definable.

The most common application of split screens in games is the status display in scrolling games. Split screens make this sort of game simpler to program because when the top half window is scrolled the programmer does not have to worry about redrawing the bottom half.

WHAT IS RLE?
RLE stands for RUN LENGTH ENCODING. It is a quick simple data compression scheme which is commonly used for image data compression or compression of any data. Although not the most efficient system, it is fast, which is why it is used in image storage systems like PCX. This implementation is more efficient than the one used in PCX files because it uses 1 bit to identify a Run Length byte as opposed to two in PCX files, but more on this later.

This set of functions can be used to implement your own compressed image file format or for example compress game maps for various levels etc. The uses are limited by your imagination. I opted for trading off PCX RLE compatibility for the improved compression efficiency.

Here is how the data is un-compressed to give an idea of its structure.

STEP 1 read a byte from the RLE compressed source buffer.

STEP 2 If has its high bit is set then the lower 7 bits represent the number of times the next byte is to be repeated in the destination buffer. If the count (lower 7 bits) is zero then we have finished decoding goto STEP 5 else goto STEP 4

STEP 3 Read a data from the source buffer and copy it directly to the destination buffer. goto STEP 1

STEP 4 Read a data byte from the source buffer and copy it to the destination buffer the number of times specified by step 2. goto STEP 1

STEP 5 Stop, decoding done.

If the byte does not have the high bit set then the byte itself is transferred to the destination buffer.
Data bytes that have the high bit already set and are unique in the input stream are represented as a Run Length of 1 (i.e. 81 which includes high bit) followed by the data byte.

If your original uncompressed data contains few consecutive bytes and most have high bit set (i.e. have values > 127) then your so called compressed data would require up to 2x the space of the uncompressed data, so be aware that the compression ratio is extremely variable depending on the type of data being compressed.

Apologies for this poor attempt at a description, but you can look up RLE in any good text. Alternatively, any text that describes the PCX file structure in any depth should have a section on RLE compression.

WHAT IS DOUBLE BUFFERING ?

Double buffering (also known as page flipping) is the technique most often used to do animation. it requires hardware that is capable of displaying multiple graphics pages (or at least 2). Animation is achieved by drawing an image in the non visible screen and then displaying the non visible screen. Once the page has been flipped the process starts again. The next frame of the animation is drawn on the non visible screen, the page is flipped again etc.

WHAT IS TRIPLE BUFFERING?

WHAT IS A BEZIER CURVE?

The Bezier curve was developedd by the French mathematician Pierre Bezier for the design of automobiles. This curve is defined by four points: two end points and two control points. The curve begins at one end point, ends at the other end point, and uses the two control points to determine the curvature of the line.

The Care and Feeding of Compiled Masked Blits

by Matthew MacKenzie

The XCBITMAP module is small, containing only three procedures:

x_compile_bitmap - Compiles your bitmap into native code which writes to the VGA screen in an X mode.

x_put_cbitmap - Converts X and Y coordinates into a location on the screen, sets up the necessary VGA registers, and executes the compiled bitmap as a subroutine.

x_sizeof_cbitmap - Takes a planar bitmap and returns an integer equal to the size of the compiled bitmap which the planar bitmap would produce. It is essentially a lobotomized version of ***x_compile_bitmap***, with all the code generation replaced with a size counter.

x_compile_bitmap scans through a source bitmap and generates 8086 instructions to plot every non-zero pixel. It is designed to be used before the action begins rather than on-the-fly. The compiled bitmap contains no branches, and no reference to the zero (transparent) pixels. Where two pixels are exactly four columns apart they are plotted with a single 16-bit store, and the VGA MAP_MASK register will be set at most four times. As a result your bitmap may run several times faster than a traditional memory-to-VGA masked blit routine. There is no way to perform clipping on these bitmaps, or to plot a pixel of color zero.

x_compile_bitmap works with bitmaps in the standard Xlib planar bitmap format. On a time scale of 60 frames per second, it is actually relatively slow. Since a compiled bitmap is relocatable you may just want to have it saved to disk, and not include the source bitmap in your program at all.

The source bitmap format is an array of bytes, a little like this:

```
char eye[] ={4, 7,   /* four byte columns across, seven rows tall */
        0, 0, 0, 0, 9, 1, 1, 1, 9, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 9, 9, 1, 1, 1, 4, 4, 9, 9, 0, 0, 0, 0, 0,
        0, 9, 9, 1, 2, 0, 0, 4, 4, 1, 9, 9, 0, 0, 0, 0,
        9, 9, 9, 1, 0, 0, 0, 0, 1, 1, 9, 9, 9, 0, 0, 0,
        0, 9, 9, 1, 2, 0, 0, 2, 1, 1, 9, 9, 0, 0, 0, 0,
        0, 0, 9, 9, 1, 1, 1, 1, 1, 9, 9, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 9, 1, 1, 1, 9, 0, 0, 0, 0, 0, 0, 0};
```

This is actually a linear bitmap, which is the wrong format for compilation, but is easier on human eyes. Use the module XBMTOOLS to convert linear bitmaps into planar bitmaps, and vice-versa.

To compile this image for a mode 360 pixels (90 byte columns) across:

```
char planar_eye[4*7 + 2];
char far * EyeSize;
```

```
(void) x_bm_to_pbm((char far *) eye, (char far *) planar_eye);
EyeSize = x_sizeof_cbitmap((far char *) planar_eye);
CompiledEye = farmalloc(EyeSize);
(void) x_compile_bitmap(90, (far char *) planar_eye, CompiledEye);
```

Notice that both buffers must exist beforehand. Since ***x_compile_bitmap*** returns the size of the compiled code, in bytes, you can reallocate the bitmap immediately to the right size if using ***x_sizeof_cbitmap*** seems inconvenient (reallocation may even be faster, though using the function is cleaner). The pointers are 32-bit because compiled bitmaps take so much space: they are at one end of the speed-versus-memory spectrum. A good rule of thumb is to allocate (3.5 * buffer-height * buffer-width) + 25 bytes (rounding up), then pare your bitmap down when you find out how much space you've actually used.

Since the compiled bitmap has to fit within one segment of memory, it cannot contain more than about 19,000 pixels. This will not be a limitation for most sane programmers. If you are not a sane programmer try splitting your huge, unwieldy image up into smaller parts. You can use the same gigantic bitmap if you divide it into horizontal slices for compilation. For that matter, dividing the source up that way will let you use a source bitmap large than 64K, which is an even sicker idea.

Back to business. A bitmap is compiled for only one width of screen. If you are using a logical screen larger than your physical screen, call the bitmap compiler with the logical width. The important thing is the number of bytes per line. Notice that you do not have to be in a graphics mode to use this routine.This allows you to develop and compile bitmaps separately, with whatever utility programs you might cook up.

The final function is ***x_put_cbitmap***.
To plot our eye at (99, 4), on the page which starts at location 0:

```
x_put_cbitmap(99, 4, 0, CompiledEye);
```

This function depends on the global variable *ScrnLogicalByteWidth* from the module XMAIN, which should be the same number as the column parameter you used to compile your bitmap.

The XCBITMAP module supports memory-to-VGA blits only. XLIB also includes non-masking routines which can quickly save and restore the background screen behind your bitmap, using fast string operations.

This module is part of the XLIB package, and is in the public domain. If you write something which uses it, though, please send me a copy as a courtesy if for no other reason so I can tilt my chair back and reflect that it may have been worth the trouble after all.

The included program DEMO2.C demonstrates the performance difference between planar bitmap masked blits and compiled bitmap blits.

Blits and Pieces
by Matthew MacKenzie

The XCLIPPBM module contains clipping versions of two of the three routines in the XPBITMAP module:

x_clip_pbm - transfers a planar bitmap to the screen, clipping off any part outside a bounding box.
x_clip_masked_pbm - does the same thing, but transfers only non-zero pixels.

The planar bitmap format is described elsewhere. Here we will look at the clipping itself, since it is the only distinguishing feature of this module.

The bounding box is made up of four integers, *TopBound, BottomBound, LeftBound,* and *RightBound*. Unlike most global variables in XLIB, these are meant to be written to. In fact, they start out uninitialized. Be sure to set them before you try plotting any clipped bitmaps.
Note that these are not the same variables which are used in the other clipping modules in XLIB. This is because the two systems are incompatible: the other modules clip horizontally to columns while this one clips to pixels. As you might have guessed, those functions and these were developed in different hemispheres of the planet. If it's any consolation, this does give you two independent bounding boxes to futz with, should the mood visit you.

Bitmaps cannot go outside the perimeter of the bounding box, but they can overlap it. If *TopBound* equals *BottomBound*, for example, a horizontal slice of a bitmap may still be plotted. It is safe to turn the box "inside out" to make sure nothing will be plotted. This is the first thing each routine checks for.

To plot a bitmap, minus its zero pixels, minus anything outside the bounding box:

x_clip_masked_pbm (int X, int Y, int ScreenOffs, char far * Bitmap);

The arguments are in the same order as those for ***x_put_masked_pbm*** in the module XPBITMAP. The bounding box is relative to the given ScreenOffs). This lets you perform page flipping without worrying about what screen you are clipping to it's always the current screen. The bitmap itself, of course, is not affected; clipping is performed on-the-fly. Both functions return an integer which indicates whether any part of the bitmap was inside the bounding box. If the entire bitmap was outside, a 1 is returned; otherwise, a 0.

The third function in XPBITMAP, for which this module has no equivalent, copies from video RAM to system RAM. The absence of such a routine may seem at first like a disadvantage but this, like so many things in this life, is an illusion. You can use the unclipped routine, and clip the bitmap when you want to plot it back onto the screen.

Wheel Have to See About That
by Matthew MacKenzie

The XCIRCLE module contains two functions, neither of which should be a big mystery:

x_circle, oddly enough, draws a circle.

x_filled_circle does too, only the circle is filled (in some libraries this is called a disc).

The word `circle' here refers to a round thing which is as many pixels tall as across. It only looks like a circle in 320x240 mode, the original mode X, and in 376x282 mode. In both functions, the circle is specified by the coordinates of the upper-left-hand corner of the smallest box which holds it, and the diameter. Some circle functions have you specify a center point; this system is kind of odd because a circle with an even diameter does not have a particular pixel for a center. Every circle, on the other hand, has a box with an upper-left corner. No bounds are checked. A diameter of zero will draw nothing, and a negative diameter will blow your VGA board into hundreds of thousands of tiny little smoldering fragments. Neither function supports clipping. The calculation of the circle is based on an algorithm described by Michael P. Lindner in a letter to the editor on page 8 of Dr. Dobb's Journal #169 (October 1990). The algorithm has been rearranged to allow drawing and moving the plots in the eight octants to be performed in one step, so that each pixel does not have to be loaded into the CPU twice. ***x_filled_circle*** does not take advantage of this optimization because it handles different parts of each plot at different times.