Libraries

COLLABORATORS

	TITLE : Libraries		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Libraries

1.1   Amiga® RKM Libraries: 30 Layers Library

This chapter describes the layers library which provides routines that are
used to manage overlapping rectangular drawing areas that share a common
display. Intuition uses the layers library to manage its system of
windows.

The chapter also describes the use of regions, special structures used to
mask off areas where drawing can take place. Regions are installed
through the layers library function InstallClipRegion() but the routines
for the creation, disposal and manipulation of regions are part of the
graphics library.

 Layers   Regions   Function Reference

1.2   30 Layers Library / Layers

The concept of a layer is closely tied to Intuition windows. A layer is a
rectangular drawing area. A layer can overlap other layers and has a
display priority that determines whether it will appear in front or behind
other layers. Every Intuition window has an associated Layer structure.
Layers allow Intuition and application programs to :

   * Share a display's BitMap among various tasks in an orderly way
     by creating layers, separate drawing rectangles, within the
     BitMap.

   * Move, size or depth-arrange a layer while automatically keeping
     track of which portions of other layers are hidden or revealed
     by the operation.

   * Manage the remapping of coordinates, so the application need not
     track the layer's offset into the BitMap.

   * Maintain each layer as a separate entity, which may optionally
     have its own BitMap.

   * Automatically update same newly visible portions.

The layers library takes care of housekeeping: the low level, repetitive
tasks which are required to keep track of where to place bits. The layers
library also provides a locking mechanism which coordinates display
updating when multiple tasks are drawing graphics to layers. The
windowing environment provided by the Intuition library is largely based
on layers.

    WARNING:

    Layers may not be created or used directly with Intuition screens.
    Intuition windows are the only supported method of adding layers to
    Intuition screens. Only the layer locking and unlocking functions
    are safe to use with Intuition. An application must create and
    manage its own View if it will be creating layers directly on the
    display.

 The Layer Structure   Working With Existing Layers
 The Layer's RastPort   Creating and Using New Layers
 Types of Layers   Layers Example
 Opening the Layers Library

1.3 30 / Layers / The Layer Structure

The internal representation of layers is essentially a set of clipping
rectangles. Each layer is represented by an instance of the Layer
structure. All the layers in a display are linked together through the
Layer_Info structure. Any display shared by multiple layers (such as an
Intuition screen) requires one Layer_Info data structure to handle
interactions between the various layers. Here is a partial listing of the
Layer structure from <graphics/clip.h>. (For a complete listing refer to
the Amiga ROM Kernel Reference Manual: Includes and Autodocs.)

```
struct Layer
{
    struct  Layer *front,*back;
    struct  ClipRect    *ClipRect; /* read ROMs to find 1st cliprect */
    struct  RastPort    *rp;
    struct  Rectangle  bounds;
       ...

    UWORD   Flags;                 /* obscured ?, Virtual BitMap? */
    struct  BitMap  *SuperBitMap;
       ...

    struct  Region  *DamageList;  /* list of rectangles to refresh */
                                  /* through */
};
```


    The Layer Structure is Read-Only.

    Applications should never directly modify any of the elements of the
    Layer structure. In addition, applications should only read the

front, back, rp, bounds, Flags, SuperBitMap and DamageList elements
of the Layer structure. (Some of these elements are subject to
dynamic change by the system so proper layer locking procedures must
be followed when relying on what the application has read.)

1.4   30 / Layers / The Layer's RastPort

When a layer is created, a RastPort is automatically to go along with it.
The pointer to the RastPort is contained in the layer data structure.
Using this RastPort, the application may draw anywhere into the layer's
bounds rectangle. If the application tries to draw outside of this
rectangle, the graphics routines will clip the graphics.

Here is sample code showing how to access the layer's RastPort:

```
struct RastPort *myRPort;
    /* allocate a RastPort pointer for each layer */

myRPort = layer->rp;

/* The layer's RastPort may be used with any of the graphics library
** calls that require this structure.  For instance, to fill layer
** with color:
*/
SetRast(layer->rp, color);

/* set up for writing text into layer */
SetDrMd(layer->rp,JAM1);
SetAPen(layer->rp,0);
Move(layer->rp, 5, 7);

/* write into layer */
Text(layer->rp, string, strlen(string));
```

1.5   30 / Layers / Types of Layers

The layers library supports three types of layers: simple refresh, smart
refresh and super bitmap. The type of the layer, specified by the Flags
field in the Layers structure, determines what facilities the layer
provides.

```
Use Only One Layer Type Flag
----------------------------
The three layer-type Flags are mutually exclusive.  That is, only one
layer-type flag (LAYERSIMPLE, LAYERSMART and LAYERSUPER) should be
specified.
```

```
 Simple Refresh Layer     Super Bitmap Layer
 Smart Refresh Layer      Backdrop Layer
```

1.6   30 / / **Types of Layers** / **Simple Refresh Layer**

```
When an application draws into the layer, any portion of the layer that is
visible (not obscured) will be rendered into the common BitMap of the
viewing area.  All graphics rendering routines are "clipped", so that only
exposed sections of the layer are drawn into.  No back-up of obscured
areas is provided.
```

```
If another layer operation is performed that causes an obscured part of a
simple refresh layer to be exposed, the application must determine if the
section need be refreshed, re-drawing the newly exposed part of the layer
as required.
```

```
The basic advantage of simple refresh is that it does not require back-up
area to save drawing sections that cannot be seen, saving memory. However,
the application needs to monitor the layer to see if it needs refreshing.
This is typically performed with statements like:
```

```
    if (layer->Flags & LAYERREFRESH)
        refresh(layer);
```

1.7   30 / / **Types of Layers** / **Smart Refresh Layer**

```
Under smart refresh, the system provides dynamic backup of obscured
sections of the layer. The graphics routines will automatically draw into
these backup areas when they encounter an obscured part of the layer. The
backup memory will be used to automatically update the display when
obscured sections later become exposed.
```

```
With smart refresh layers, the system handles all of the refresh
requirements of the layer, except when the layer is made larger. When
parts of the layer are exposed by a sizing operation, the application must
refresh the newly exposed areas.
```

```
The advantage of smart refresh is the speed of updating exposed regions of
the layer and the ability of the system to manage part of the updating
process for the application.. Its main disadvantage is the additional
memory required to handle this automatic refresh.
```

1.8   30 / / **Types of Layers** / **Super Bitmap Layer**

```
A super bitmap layer is similar to a smart refresh layer.  It too has a
back-up area for rendering graphics for currently obscured parts of the
display.  Whenever an obscured area is made visible, the corresponding
part of the backup area is copied to the display automatically.
```

```
However, it differs from smart refresh in that:
```

```
  * The back-up BitMap is user-supplied, rather than being allocated
    dynamically by the system.
```

```
  * The back-up BitMap may be as large or larger than the the current
```

        size of the layer. It may also be larger than the maximum size of the
        layer.

To see a larger portion of a super bitmap on-display, use SizeLayer(). To
see a different portion of the super bitmap in the layer, use
ScrollLayer().

When the graphics routines perform drawing commands, part of the drawing
appears in the common BitMap (the on-display portion). Any drawing outside
the displayed portion itself is rendered into the super bitmap. When the
layer is scrolled or sized, the layer contents are copied into the super
bitmap, the scroll or size positioning is modified, and the appropriate
portions are then copied back into the layer. (Refer to the graphics
library functions SyncSBitMap() and CopySBitMap().)

1.9   30 / / Types of Layers / Backdrop Layer

A layer of any type may be designated a backdrop layer which will always
appear behind all other layers. They may not be moved, sized, or
depth-arranged. Non-backdrop layers will always remain in front of
backdrop layers regardless of how the non-backdrop layer is moved, sized
or depth-arranged.

1.10   30 / Layers / Opening the Layers Library

Like all libraries, the layers library must be opened before it may be
used. Check the Layers Autodocs to determine what version of the library
is required for any particular Layers function.

```
    struct Library *LayersBase;

    if (NULL != (LayersBase = OpenLibrary("layers.library",33L)))
        {
        /* use Layers library */

        CloseLibrary((struct Library *)LayersBase);
        }
```


1.11   30 / Layers / Working With Existing Layers

A common operation performed by applications is to render text or graphics
into an existing layer such as an Intuition window. To prevent Intuition
from changing the layer (for instance when the user resizes or moves the
window) during a series of graphic operations, the layers library provides
locking functions for obtaining exclusive access to a layer.

These locking functions are also useful for applications that create their
own layers if the application has more than one task operating on the
layers asynchronously. These calls coordinate multiple access to layers.

```
Table 30-1: Functions for Intertask Control of Layers (Layers Library)
 _____
|                                                                   |
|        LockLayer()   Lock out rendering in a single layer.        |
|      UnlockLayer()   Release LockLayer() lock.                    |
|       LockLayers()   Lock out rendering in all layers of a display. |
|     UnlockLayers()   Release LockLayers() lock.                   |
|    LockLayerInfo()   Gain exclusive access to the display's layers. |
|  UnlockLayerInfo()   Release LockLayerInfo() lock.                |
|_____|
```

The following routines from the graphics library also allow multitasking
access to layer structures:

```
Table 30-2: Functions for Intertask Control of Layers (Graphics Library)
 _____
|                                                                   |
|         LockLayerRom()   Same as LockLayer(), from layers library. |
|       UnlockLayerRom()   Release LockLayerRom() lock.             |
|  AttemptLockLayerRom()   Lock layer only if it is immediately     |
|                          available.                               |
|_____|
```

These functions are similar to the layers LockLayer() and UnlockLayer()
functions, but do not require the layers library to be open. See the
Amiga ROM Kernel Reference Manual: Includes and Autodocs for details.

  Intertask Operations   Determining Layer Position

1.12   30 / / Working With Existing Layers / Intertask Operations

If multiple tasks are manipulating layers on the same display they will be
sharing a Layer_Info structure and their use of it and its related data
structures need to be coordinated. To ensure that a structure remains
cohesive, it should be operated on by only one task at a time. The
Layer_Info encompasses all the layers existing on a single display.

LockLayerInfo() must be called whenever the visible portions of layers may
be affected, or when the Layer_Info structure is changed.

```
    void LockLayerInfo( struct Layer_Info *li );
```

The lock should be obtained whenever a layer is created, deleted sized or
moved, as the list of layers that is being managed by the Layer_Info data
structure must be updated.

It is not necessary to lock the Layer_Info data structure while rendering,
or when calling routines like ScrollLayer(), because layer sizes and
on-display positions are not being affected.

Use UnlockLayerInfo() when you have finished the layer operation:

```
    void UnlockLayerInfo( struct Layer_Info *li );
```

If you don't unlock the Layer_Info then any other task calling
LockLayerInfo() on the same Layer_Info structure will be blocked creating
a potential deadlock situation.

In addition to locking the Layer_Info structure, the layer itself should
be locked if it is shared between tasks so that only one task at a time
renders graphics to it. LockLayer() is used to get exclusive graphics
output to a layer.

```
    void LockLayer( long dummy, struct Layer *layer );
```

If a graphics function is in process, the lock will return when the
function is completed. Other tasks are blocked only if they attempt to
draw graphics into this layer, or try to obtain a lock on this layer. The
MoveLayer(), SizeLayer() and ScrollLayer() functions automatically lock
and unlock the layer they operate on.

UnlockLayer() should be used after the graphics operation to make the
layer available to other tasks again.

```
    void UnlockLayer( struct Layer *layer );
```

If more than one layer must be locked, then the LockLayer() calls should
be surrounded by LockLayerInfo() and UnlockLayerInfo() calls, to prevent
deadlock situations.

The layers library provides two additional functions, LockLayers() and
UnlockLayers(), for locking multiple layers.

```
    void LockLayers( struct Layer_Info *li );
    void UnlockLayers( struct Layer_Info *li );
```

LockLayers() is used to lock all layers in a single command.
UnlockLayers() releases the layers lock. The system calls these routines
during the BehindLayer(), UpfrontLayer() and MoveLayerInFrontOf()
operations (described below).

1.13   30 / / Working With Existing Layers / Determining Layer Position

If the viewing area has been separated into several layers, the
application may need to find out which layer is topmost at a particular
x,y coordinate. Use the WhichLayer() function for this:

```
    struct Layer *WhichLayer( struct Layer_Info *li, long x, long y );
```

To be sure that no task adds, deletes, or changes the sequence of layers
before this information is used, call LockLayerInfo() before calling
WhichLayer(), and call UnlockLayerInfo() when the operation is complete.
In this way, the program may ensure that it is acting on valid
information. Always check for a NULL return value (coordinate not in a
layer) from WhichLayer().

1.14   30 / Layers / Creating and Using New Layers

The functions described in this section are generally not safe to use with
Intuition. To create new layers for Intuition you use Intuition window
calls (see the "Intuition Windows" chapter earlier in this book).

Only applications that create and mange their own View will be able to
call the layer creation and updating functions discussed here.

```
            Table 30-3: Functions for Creating and Updating Layers
        _____
       |                                                                |
       |            NewLayerInfo()  Allocating a Layer_Info structure.   |
       |        DisposeLayerInfo()  Deallocating a Layer_Info structure. |
       |----------------------------------------------------------------|
       |       CreateUpfrontLayer()  Make a new layer in front of others. |
       |       CreateBehindLayer()  Make a new layer behind others.      |
       |             DeleteLayer()  Remove and delete an existing layer. |
       |----------------------------------------------------------------|
       |              MoveLayer()  Change the position (not depth) of a  |
       |                            layer.                               |
       |              SizeLayer()  Change the size of a layer.           |
       |            ScrollLayer()  Change the internal coordinates of a  |
       |                            layer.                               |
       |----------------------------------------------------------------|
       |            BehindLayer()  Depth arrange a layer behind others.  |
       |           UpfrontLayer()  Depth arrange a layer in front of     |
       |                            others.                              |
       |      MoveLayerInFrontOf()  Depth arrange a layer to a specific  |
       |                            position.                            |
       |----------------------------------------------------------------|
       |  SwapBitsRastPortClipRect()  Fast, non-layered and non-damaging  |
       |                            display operation.                   |
       |----------------------------------------------------------------|
       |            BeginUpdate()  Synchronize optimized refreshing for  |
       |                            layer.                               |
       |              EndUpdate()  End optimized layer refresh.          |
       |_____|
```

```
Creating a Viewing Workspace
Creating the Layers
Allocating and Deallocating Layer_Info
Allocating and Deallocating Layers
Moving and Sizing Layers
Changing a Viewpoint
Reordering Layers
Sub-Layer Rectangle Operations
```

1.15   30 / / Creating and Using New Layers / Creating a Viewing Workspace

A viewing workspace may be created by using the primitives InitVPort(),
InitView(), MakeVPort(), MrgCop(), and LoadView(). Please reference the

"Graphics Primitives" chapter for details on creating a low-level graphics
display. Do not create Layers directly on Intuition screens. Windows are
the only supported method of creating a layer on a screen.

1.16   30 / / **Creating and Using New Layers** / **Creating the Layers**

The application must first allocate and initialize a Layer_Info data
structure which the system uses to keep track of layers that are created,
use statements like:

```
struct Layer_Info *theLayerInfo;

if (NULL != (theLayerInfo = NewLayerInfo()))
    {
    /* use Layer_Info */

    DisposeLayerInfo(theLayerInfo);
    }
```

Layers may be created in the common bit map by calling
CreateUpfrontLayer() or CreateBehindLayer(), with a sequence such as the
following:

```
struct Layer      *layer;
struct Layer_Info *theLayerInfo;
struct BitMap     *theBitMap;

/* requests construction of a smart refresh layer. */
if (NULL == (layer = CreateUpfrontLayer(theLayerInfo, theBitMap,
        20, 20, 100, 80, LAYERSMART, NULL)))
    error("CreateUpfrontLayer() failed.");
else
    {
    ; /* layer successfully created here. */
    }
```

1.17   30 / / / **Allocating and Deallocating Layer_Info**

Use NewLayerInfo() to allocate and initialize a Layer_Info structure and
associated sub-structures.

```
struct Layer_Info *NewLayerInfo( void );
```

You must call this function before attempting to use any of the other
layers functions described below. When you have finished with a
Layer_Info structure, use DisposeLayerInfo() to deallocate it.

```
void DisposeLayerInfo( struct Layer_Info *li );
```

This function deallocates a Layer_Info and associated structures
previously allocated with NewLayerInfo().

1.18   30 / / Creating and Using Layers / Allocating and Deallocating Layers

Layers are created using the routines CreateUpfrontLayer()
and CreateBehindLayer(). CreateUpfrontLayer()
creates a layer that will appear in front of any existing layers.

```
struct Layer *CreateUpfrontLayer( struct Layer_Info *li,
                                   struct BitMap *bm,
                                   long x0, long y0, long x1, long y1,
                                   long flags, struct BitMap *bm2 );
```

CreateBehindLayer() creates a layer that appears behind existing layers,
but in front of backdrop layers.

```
struct Layer *CreateBehindLayer( struct Layer_Info *li,
                                  struct BitMap *bm,
                                  long x0, long y0, long x1, long y1,
                                  long flags, struct BitMap *bm2 );
```

Both of these routines return a pointer to a Layer data structure (as
defined in the include file <graphics/layers.h>), or NULL if the operation
was unsuccessful.

```
A New Layer Also Gets a RastPort.
---------------------------------
When a layer is created, the routine automatically creates a
RastPort to go along with it.  If the layer's RastPort is passed
to the drawing routines, drawing will be restricted to the layer.
See "The Layer's RastPort" section above.
```

Use the DeleteLayer() call to remove a layer:

```
LONG DeleteLayer( long dummy, struct Layer *layer );
```

DeleteLayer() removes a layer from the layer list and frees the memory
allocated by the layer creation calls listed above.

1.19   30 / / Creating and Using New Layers / Moving and Sizing Layers

The layers library includes three functions for moving and sizing layers:

```
LONG MoveLayer( long dummy, struct Layer *layer, long dx, long dy );
LONG SizeLayer( long dummy, struct Layer *layer, long dx, long dy );
LONG MoveSizeLayer( struct Layer *layer, long dx, long dy, long dw,
                    long dh);
```

MoveLayer() moves a layer to a new position relative to its current
position. SizeLayer() changes the size of a layer by modifying the
coordinates of the lower right corner of the layer. MoveSizeLayer()
changes both the size and position of a layer in a single call.

1.20   30 / / **Creating and Using New Layers / Changing a Viewpoint**

The ScrollLayer() function changes the portion of a super bitmap that is
shown by a layer:

```
    void ScrollLayer( long dummy, struct Layer *layer, long dx, long dy );
```

This function is most useful with super bitmap layers but can also
simulate the effect on other layer types by adding the scroll offset to
all future rendering.

1.21   30 / / **Creating and Using New Layers / Reordering Layers**

The layers library provides three function calls for reordering layers:

```
    LONG BehindLayer ( long dummy, struct Layer *layer );
    LONG UpfrontLayer( long dummy, struct Layer *layer );
    LONG MoveLayerInFrontOf( struct Layer *layer_to_move,
                             struct Layer *other_layer );
```

BehindLayer() moves a layer behind all other layers. This function
considers any backdrop layers, moving a current layer behind all others
except backdrop layers. UpfrontLayer() moves a layer in front of all
other layers. MoveLayerInFrontOf() is used to place a layer at a specific
depth, just in front of a given layer.

As areas of simple refresh layers become exposed, due to layer movement or
sizing for example, the newly exposed areas have not been drawn into, and
need refreshing. The system keeps track of these areas by using a
DamageList. To update only those areas that need it, the BeginUpdate()
EndUpdate() functions are called.

```
    LONG BeginUpdate( struct Layer *l );
    void EndUpdate  ( struct Layer *layer, unsigned long flag );
```

BeginUpdate() saves the pointer to the current clipping rectangles and
installs a pointer to a set of ClipRects generated from the DamageList in
the layer structure. To repair the layer, use the graphics rendering
routines as if to redraw the entire layer, and the routines will
automatically use the new clipping rectangle list. So, only the damaged
areas are actually rendered into, saving time.

```
    Never Modify the DamageList.
    ----------------------------
    The system generates and maintains the DamageList region.  All
    application clipping should be done through the InstallClipRegion()
    function.
```

To complete the update process call EndUpdate() which will restore the
original ClipRect list.

1.22   30 / / Creating and Using New Layers / Sub-Layer Rectangle Operations

The SwapBitsRastPortClipRect() routine is for applications that do not
want to worry about clipping rectangles.

```
    void SwapBitsRastPortClipRect( struct RastPort *rp,
                                   struct ClipRect *cr );
```

For instance, you may use The SwapBitsRastPortClipRect() to produce a menu
without using Intuition. There are two ways to produce such a menu:

 1. Create an up-front layer with CreateUpfrontLayer(), then render the
    menu in it. This could use lots of memory and require a lot of (very
    temporary) "slice-and-dice" operations to create all of the clipping
    rectangles for the existing windows and so on.

 2. Use SwapBitsRastPortClipRect(), directly on the display drawing area:

     * Render the menu in a back-up area off the display, then lock all
       of the on-display layers so that no task may use graphics
       routines to draw over the menu area on the display.

     * Next, swap the on-display bits with the off-display bits, making
       the menu appear.

     * When finished with the menu, swap again and unlock the layers.

The second method is faster and leaves the clipping rectangles and most of
the rest of the window data structures untouched.

```
    Warning:
    --------
    All of the layers must be locked while the menu is visible if you
    use the second method above.  Any task that is using any of the
    layers for graphics output will be halted while the menu operations
    are taking place.  If, on the other hand, the menu is rendered as a
    layer, no task need be halted while the menu is up because the lower
    layers need not be locked.
```

1.23   30 Layers Library / Regions

Regions allow the application to install clipping rectangles into layers.
A clipping rectangle is a rectangular area into which the graphics
routines will draw. All drawing that would fall outside of that
rectangular area is clipped (not rendered).

User clipping regions are linked lists of clipping rectangles created by
an application program through the graphics library routines described
below. By combining together various clipping rectangles, any arbitrary
clipping shape can be created. Once the region is set up, you use the
layers library call InstallClipRegion() to make the clipping region active
in a layer.

Regions are safe to use with layers created by Intuition (i.e., windows).

The following table describes the routines available for the creation,
manipulation and use of regions.

Table 30-4: Functions Used with Regions

Routine	Library	Description
InstallClipRegion()	Layers	Add a clipping region to a layer.
NewRegion()	Graphics	Create a new, empty region.
DisposeRegion()	Graphics	Dispose of an existing region and its rectangles.
AndRectRegion()	Graphics	And a rectangle into a region.
OrRectRegion()	Graphics	Or a rectangle into a region.
XorRectRegion()	Graphics	Exclusive-or a rectangle into a region.
ClearRectRegion()	Graphics	Clear a rectangular portion of a region.
AndRegionRegion()	Graphics	And two regions together.
OrRegionRegion()	Graphics	Or two regions together.
XorRegionRegion()	Graphics	Exclusive-or two regions together.
ClearRegion()	Graphics	Clear a region.

With these functions, the application can selectively update a
custom-shaped part of a layer without disturbing any of the other layers
that might be present.

Never Modify the DamageList of a Layer Directly.

Use the routine InstallClipRegion() to add clipping to the layer.
The regions installed by InstallClipRegion() are independent of
the layer's DamageList and use of user clipping regions will not
interfere with optimized window refreshing.

Do Not Modify A Region After It Has Been Added.

After a region has been added with InstallClipRegion(), the
program may not modify it until the region has been removed with
another call to InstallClipRegion().

Creating and Deleting Regions   Changing a Region
Installing Regions   Regions Example

1.24   30 / Regions / Creating and Deleting Regions

You allocate a Region data structure with the NewRegion() call.

```
struct Region *NewRegion( void );
```

The NewRegion() function allocates and initializes a Region structure that
has no drawable areas defined in it. If the application draws through a
new region, nothing will be drawn as the region is empty. The application
must add rectangles to the region before any graphics will appear.

Use DisposeRegion() to free the Region structure when you are done with it.

```
void DisposeRegion( struct Region *region );
```

DisposeRegion() returns all memory associated with a region to the system
and deallocates all rectangles that have been linked to it.

```
Don't Forget to Free Your Rectangles.
-------------------------------------
All of the functions that add rectangles to the region make copies
of the rectangles.  If the program allocates a rectangle, then adds
it to a region, it still must deallocate the rectangle.  The call to
DisposeRegion() will not deallocate rectangles explicitly
allocated by the application.
```

1.25 30 / Regions / Installing Regions

Use the function InstallClipRegion() to install the region.

```
struct Region *InstallClipRegion( struct Layer *layer,
                                  struct Region *region );
```

This installs a transparent clipping region to a layer. All subsequent
graphics calls will be clipped to this region. The region must be removed
with a second call to InstallClipRegion() before removing the layer.

```
/*
** Sample installation and removal of a clipping region
*/
register struct Region     *new_region ;
register struct Region     *old_region ;

/* If the application owns the layer and has not installed a region,
** old_region will return NULL here.
*/
old_region = InstallClipRegion(win->WLayer, new_region);

/* draw into the layer or window */

if (NULL != (old_region = InstallClipRegion(win->WLayer, old_region)))
    {
    /* throw the used region away.  This region could be saved and
    ** used again later, if desired by the application.
    */
    DisposeRegion(new_region) ;
    }


A Warning About InstallClipRegion().
-------------------------------------
```

The program must not call InstallClipRegion() inside of a
Begin/EndRefresh() or Begin/EndUpdate() pair. The following code
segment shows how to modify the user clipping region when using these
calls. See the Autodoc for BeginRefresh() for more details.


```
register struct Region      *new_region ;
register struct Region      *old_region ;

/* you have to have already setup the new_region and old_region */

BeginRefresh(window);
/* draw through the damage list */
/* into the layer or window */
EndRefresh(window, FALSE);                    /* keep the damage list */

old_region = InstallClipRegion(win->WLayer, new_region);

BeginRefresh(window);
/* draw through the damage list and the new_region */
/* into the layer or window */
EndRefresh(window, FALSE);                    /* keep the damage list */

/* put back the old region */
new_region = InstallClipRegion(win->WLayer, old_region);

BeginRefresh(window);
EndRefresh(window, TRUE);                      /* remove the damage list */

old_region = InstallClipRegion(win->WLayer, new_region);

BeginRefresh(window);
/* draw through the new_region only into the layer or window */
EndRefresh(window, FALSE);

/* finally get rid of the new region, old_region still installed */
if (NULL != (new_region = InstallClipRegion(win->WLayer, old_region)))
    DisposeRegion(new_region) ;
```

1.26   30 / Regions / Changing a Region

Regions may be modified by performing logical operations with rectangles,
or with other regions.

```
Reuse Your Rectangles.
----------------------
In all of the rectangle and region routines the clipping rectangle
is copied into the region.  This means that a single clipping
rectangle (Rectangle structure) may be used many times by simply
changing the x and y values.  The application need not create a
new instance of the Rectangle structure for each rectangle added
to a region.
```

For instance:

```
    extern struct Region *RowRegion;  /* created elsewhere */

    WORD ktr;
    struct Rectangle rect;

    for (ktr = 1; ktr < 6; ktr++)
        {
        rect.MinX = 50;
        rect.MaxX = 315;
        rect.MinY = (ktr * 10) - 5;
        rect.MaxY = (ktr * 10);

        if (!OrRectRegion(RowRegion, &rect))
            clean_exit(RETURN_WARN);
        }
```

 Rectangles and Regions   Regions and Regions

1.27   30 / / Changing a Region / Rectangles and Regions

There are four functions for changing a region through logical operations
with a rectangle.

```
    BOOL OrRectRegion   ( struct Region *region,
                          struct Rectangle *rectangle );
    void AndRectRegion  ( struct Region *region,
                          struct Rectangle *rectangle );
    BOOL XorRectRegion  ( struct Region *region,
                          struct Rectangle *rectangle );
    BOOL ClearRectRegion( struct Region *region,
                          struct Rectangle *rectangle );
```

OrRectRegion() modifies a region structure by or'ing a clipping rectangle
into the region. When the application draws through this region (assuming
that the region was originally empty), only the pixels within the clipping
rectangle will be affected. If the region already has drawable areas, they
will still exist, this rectangle is added to the drawable area.

AndRectRegion() modifies the region structure by and'ing a clipping
rectangle into the region. Only those pixels that were already drawable
and within the rectangle will remain drawable, any that are outside of it
will be clipped in future.

XorRectRegion() applies the rectangle to the region in an exclusive-or
mode. Within the given rectangle, any areas that were drawable become
clipped, any areas that were clipped become drawable. Areas outside of
the rectangle are not affected.

ClearRectRegion() clears the rectangle from the region. Within the given
rectangle, any areas that were drawable become clipped. Areas outside of
the rectangle are not affected.

1.28 30 / / Changing a Region / Regions and Regions

As with rectangles and regions, there are four layers library functions
for combining regions with regions:

```
    BOOL AndRegionRegion( struct Region *srcRegion,
                          struct Region *destRegion );
    BOOL OrRegionRegion ( struct Region *srcRegion,
                          struct Region *destRegion );
    BOOL XorRegionRegion( struct Region *srcRegion,
                          struct Region *destRegion );
    void ClearRegion    ( struct Region *region );
```

AndRegionRegion() performs a logical and operation on the two regions,
leaving the result in the second region. The operation leaves drawable
areas wherever the regions drawable areas overlap. That is, where there
are drawable areas in both region 1 and region 2, there will be drawable
areas left in the result region.

OrRegionRegion() performs a logical or operation on the two regions,
leaving the result in the second region. The operation leaves drawable
areas wherever there are drawable areas in either region. That is, where
there are drawable areas in either region 1 or region 2, there will be
drawable areas left in the result region.

XorRegionRegion() performs a logical exclusive-or operation on the two
regions, leaving the result in the second region. The operation leaves
drawable areas wherever there are drawable areas in either region but not
both. That is, where there are drawable areas in either region 1 or
region 2, there will be drawable areas left in the result region. But
where there are drawable areas in both region 1 and region 2, there will
not be drawable areas left in the result region.

ClearRegion() puts the region back to the same state it was in when the
region was created with NewRegion(), that is, no areas are drawable.

1.29 30 Layers Library / Function Reference

The following are brief descriptions of the layers library functions and
related routines from the graphics library. See the Amiga ROM Kernel
Reference Manual: Includes and Autodocs for details on each function call.

                    Table 30-5: Layers Library Functions

```
 _____
|                                                                     |
|           Function              Description                         |
|=====================================================================|
|           NewLayerInfo()  Allocating a Layer_Info structure.        |
|       DisposeLayerInfo()  Deallocating a Layer_Info structure.      |
|---------------------------------------------------------------------|
|      CreateUpfrontLayer()  Make a new layer in front of others.     |
|       CreateBehindLayer()  Make a new layer behind others.          |
|            DeleteLayer()  Remove and delete an existing layer.      |
```

```
|-----------------------------------------------------------------------|
|                   MoveLayer()  Change the position (not depth) of a   |
|                                layer.                                 |
|                   SizeLayer()  Change the size of a layer.            |
|                 ScrollLayer()  Change the internal coordinates of a   |
|                                layer.                                 |
|-----------------------------------------------------------------------|
|                 BehindLayer()  Depth arrange a layer behind others.   |
|                UpfrontLayer()  Depth arrange a layer in front of      |
|                                others.                                |
|           MoveLayerInFrontOf()  Depth arrange a layer to a specific   |
|                                position.                              |
|-----------------------------------------------------------------------|
|                  WhichLayer()  Find the frontmost layer at a position.|
|-----------------------------------------------------------------------|
|    SwapBitsRastPortClipRect()  Fast, non-layered and non-damaging     |
|                                display operation.                     |
|-----------------------------------------------------------------------|
|                 BeginUpdate()  Synchronize optimized refreshing for   |
|                                layer.                                 |
|                   EndUpdate()  End optimized layer refresh.           |
|-----------------------------------------------------------------------|
|                   LockLayer()  Lock out rendering in a single layer.  |
|                 UnlockLayer()  Release LockLayer() lock.              |
|                  LockLayers()  Lock out rendering in all layers of a  |
|                                display.                               |
|                UnlockLayers()  Release LockLayers() lock.             |
|                LockLayerInfo()  Gain exclusive access to the display's|
|                                layers.                                |
|              UnlockLayerInfo()  Release LockLayerInfo() lock.         |
|-----------------------------------------------------------------------|
|             InstallClipRegion()  Add a clipping region to a layer.    |
|_____|
```

The following routines from graphics library are also required for certain
layers library functions:

```
 _____
|                                                                       |
|           Routine                      Description                    |
|=======================================================================|
|           LockLayerRom()  Same as LockLayer(), from layers library.   |
|         UnlockLayerRom()  Release LockLayerRom() lock.                |
|    AttemptLockLayerRom()  Lock layer only if it is immediately        |
|                           available.                                  |
|-----------------------------------------------------------------------|
|               NewRegion()  Create a new, empty region.                |
|           DisposeRegion()  Dispose of an existing region and its      |
|                           rectangles.                                 |
|-----------------------------------------------------------------------|
|           AndRectRegion()  AND a rectangle into a region.             |
|            OrRectRegion()  OR a rectangle into a region.              |
|           XorRectRegion()  Exclusive-OR a rectangle into a region.    |
|         ClearRectRegion()  Clear a region.                            |
|         AndRegionRegion()  AND two regions together.                  |
|          OrRegionRegion()  OR two regions together.                   |
|                                                                       |
```

```
|        XorRegionRegion()   Exclusive-OR two regions together.          |
|           ClearRegion()    Clear a region.                             |
|_____|
```

```
|        XorRegionRegion()   Exclusive-OR two regions together.          |
|           ClearRegion()    Clear a region.                             |
```