4. Blob Manager Routines

Unless otherwise specified, routines that take a blob part code as a parameter may be assumed to understand what to do with any of the codes `inFullBlob`, `inDragBlob` or `inStatBlob`.

4.1 Initialization

Threre is no specific initialization call. (This would not be true if the Blob Manager were written in a language not allowing static initialization, such as Pascal). If a large number of blobs will be created, it might be a good idea for the host to call `MoreMasters()` several times (see the Memory Manager manual). In a TransSkel application, the number of times to call `MoreMasters()` can be specified in the structure passed to `SkelInit()`.

Initial values of important Blob Manager variables are:

`BlobClick()` Zoomback Flags
- Zoomback on unglue transactions `true`
- Zoomback on bad drags `true`

`BlobClick()` Transaction Permissions
- Unglue transactions allowed `true`
- Replace transactions allowed `true`
- Transfer transactions allowed `true`
- Duplication transactions allowed `false`
- Swap transactions allowed `true`

`BlobClick()` Advisory Filter none

Dragging and Tracking
- default drag axis `noConstraint`
- default drag limit rectangle standard Macintosh screen size
- default drag slop rectangle wide open rectangle

Dimming
- dimming pattern `gray`
- dimming mode `patBic`

4.2 Blob Allocation and Disposal

pascal BlobSetHandle
NewBlobSet (void);

`NewBlobSet()` creates a new, empty blob set and returns a handle to it. The blob set is added to the Blob Manager's master list of blob sets.

If insufficient memory is available, `NewBlobSet()` returns `nil`.

pascal BlobHandle
NewBlob (BlobSetHandle bSet, Boolean enable, short glueMVal,

Creates a new blob and adds it to the given set. A handle to the new blob is returned. The blob is enabled according to the value of the `enable` parameter. This does *not* draw the blob if the enable parameter is `true`, (since it has no image initially anyway), but means that the blob will be drawn as soon as an image is defined. If `enable` is `false`, the blob will not be drawn until `ShowBlob()` is called. The `glueMVal` parameter is the number of receptors to which the blob

can be glued before it goes dim. If it is equal to `infiniteGlue` (`0x8000`) the blob can be glued to any number of other blobs. If `explicit` is `true`, the blob requires an explicit match. The `refCon` reference value is stored into the blob record, and can be used for any purpose you wish.

All new blobs are given empty static and drag regions. The handles to the match set and to the glob are set to `nil`. The `glueCount` field is set to zero. The initial drawing modes of the static and drag regions are set to `ormalDraw`.

Blobs are always added to the end of their set. A handle to the *i*-th blob in a set may be obtained with `GetBlobHandle()`. The number of blobs in a set may be obtained with `BlobSetSize()`.

If insufficient memory is available to create a new blob, `NewBlob()` returns `nil`.

pascal void
NewBlobMatch (BlobHandle b1, BlobHandle b2);

`NewBlobMatch()` adds `b1` to the match set of `b2`. This means that whenever the `glob` field of `b2` is equal to `b1`, `BlobQuiet(b2)` returns `true`. (The section discussing quiet and noisy blobs should be consulted for a complete description of blob matching properties.)

If insufficient memory is available to create a new match record, `NewBlobMatch()` returns `nil`.

pascal void
DisposeBlobSets (void);

`DisposeBlobSets()` disposes of all blob sets in the master blob set list. This may be called at application exit time to shut down the Blob Manager, but that is not strictly necessary.

pascal void
DisposeBlobSet (BlobSetHandle bSet);

`DisposeBlobSet()` removes the set from the master blob set list and disposes of all the blobs in the set. The handle to the blob set, and all handles to members of the set become invalid. In particular, if any blobs in the set are glued to any blobs, the `glob` handles in the latter blobs become invalid. You should never dispose of a blob set containing blobs that have a non-zero `glueCount` field unless you are also disposing of the blobs they are glued to.

Note
`DisposeBlobSet()` does not hide blobs before disposing of them.

pascal void
ClobberBlob (BlobHandle b, BlobSetHandle bSet);

`ClobberBlob()` deletes the blob from the blob set and disposes of it. All handles to the blob become invalid. In particular, if the blob is glued to any blobs, the `glob` handles in those blobs become invalid. A blob that has a non-zero `glueCount` field should be unglued before it is deleted.

pascal void
DisposeBlob (BlobHandle b);

Disposes of the blob and any data structures owned by it. All blobs own their pictures and their match set. The blob's glob is not disposed of, since glue blobs are not owned but only associated with other blobs. Similarly, the blobs in the match set are not disposed of, but only the match records themselves (see `DisposeMatchSet()`). Any handle to the blob becomes invalid.

`DisposeBlob()` is not like `DisposeControl()`, which takes the control out of the owner window's control list. It simply disposes of the blob, and is intended for use only by the Blob Manager. To dispose of a blob and remove it from its set, use `ClobberBlob()` instead.

pascal void
DisposeBlobPic (BlobHandle b);

`DisposeBlobPic()` disposes of any picture installed by the blob and sets the picture handle in the blob record to `nil`. The blob becomes undrawable, but `DisposeBlobPic()` does not erase it. Thus `HideBlob()` should normally be called first, unless the application is in the process of shutting down.

pascal void
ClobberBlobMatch (BlobHandle b1, BlobHandle b2);

`ClobberBlobMatch()` deletes `b1` from the match set of `b2`.

pascal void
DisposeBlobMatchSet (BlobHandle b);

Disposes of the blob's match set, if it has one. This does not dispose of the blobs whose handles are stored in the match records, only the match records themselves.

4.3 Blob Record Field Access

pascal BlobHandle
BGlob (BlobHandle b);

`BGlob()` returns a handle to the blob glued to `b`.

pascal BlobHandle
FirstBlob (BlobSetHandle bSet);

`GetFirstBlob()` returns a handle to the first blob in the set.

pascal BlobHandle
LastBlob (BlobSetHandle bSet);

`GetLastBlob()` returns a handle to the last blob in the set.

pascal BlobHandle
NextBlob (BlobHandle b);

`NextBlob()` returns a handle to the next blob in the set after `b`.

pascal BlobHandle
FirstBMatch (BlobHandle b);

`FirstBMatch()` returns a handle to the first blob in the blob's match set.

pascal RgnHandle
BStatRgn (BlobHandle b);

`BStat`Rgn`()` returns the blob's static region handle.

pascal RgnHandle
BDragRgn (BlobHandle b);

`BDrag`Rgn`()` returns the blob's drag region handle.

pascal Rect
BStatBox (BlobHandle b);

`BStatBox()` returns the `rgnBBox` field of the blob's static region.

pascal Rect
BDragBox (BlobHandle b);

`BDragBox()` returns the `rgnBBox` field of the blob's drag region.

pascal long
GetBRefCon (BlobHandle b);

`GetBRefCon()` returns the reference value of the blob.

pascal void
SetBRefCon (BlobHandle b, long refCon);

`SetBRefCon()` sets the reference value of the given blob to the given value.

pascal void
SetBGlueMax (BlobHandle b, short max);

Sets the glue count of the blob to max. In subsequent transactions, the blob is dimmed and becomes unavailable for dragging if it is simultaneously glued to max blobs.

When `SetBGlueMax()` is called, if the value of max is less than or equal to the number of blobs that the blob currently is glued to, it becomes dimmed. If the blob is dimmed, and max is greater than the number of blobs the blob is glued to, the blob becomes undimmed.

pascal short
GetBGlueMax (BlobHandle b);

`GetBGlueMax()` returns the maximum glue count of the blob.

4.4 Picture Blob Picture Creation and Region Assignment

pascal void
OpenBlob (void);

`OpenBlob()` tells the Blob Manager to start saving QuickDraw calls as a picture definition. After calling `OpenBlob()` you should draw the image of the entire blob, then call `CloseRectBlob()` or `CloseRgnBlob()` to establish the static and drag regions.

The call to `OpenBlob()` must always be balanced by a call to `CloseRectBlob()` or `CloseRgnBlob()` after the blob's picture has been drawn. Since `OpenBlob()` starts saving the blob picture by calling `OpenPicture()`, you should not call `OpenBlob()` or `OpenPicture()` while another call to `OpenBlob()` or `OpenPicture()` is in progress.

pascal void
CloseRgnBlob (BlobHandle b, RgnHandle dragRgn, RgnHandle statRgn);

`CloseRgnBlob()` calls `ClosePicture()` to stop saving the picture started by `OpenBlob()`, installs it into the given blob, and installs regions into the blob corresponding to `dragRgn` and `statRgn`. The static region actually assigned is set to the difference of `statRgn` and `dragRgn`.

The bounds rectangles of the static and drag regions are installed as the blob's `statRect` and `dragRect` fields, respectively.

If the blob is enabled, `CloseRgnBlob()` draws the blob before returning.

The caller is responsible for disposing of the regions passed to `CloseRgnBlob()`; those regions are used to construct new regions which are installed into the blob.

pascal void
CloseRectBlob (BlobHandle b, Rect *dragRect, Rect *statRect);

`CloseRectBlob()` calls `ClosePicture()` to stop saving the picture started by `OpenBlob()` and installs it into the given blob. `statFrame` and `dragFrame` should be equal to the rectangles bounding the static region and the drag region. The drag region is set to be equivalent to the `dragFrame` rectangle. The static region is set to be equivalent to the `statFrame` rectangle minus any overlap with `dragFrame`.

The bounds rectangles of the static and drag regions are installed as the blob's `statRect` and `dragRect` fields, respectively.

If the blob is enabled, `CloseRectBlob()` draws the blob before returning.

4.5 Procedure Blob Procedure Installation and Region Assignment

pascal void
SetProcRgnBlob (BlobHandle b, BDrawProcPtr proc,

`SetProcRgnBlob()` installs `proc` as the blob's drawing procedure and installs regions into the blob corresponding to `dragRgn` and `statRgn`. The static region actually assigned is set to the difference of `statRgn` and `dragRgn`.

The bounding rectangles of the static and drag regions are installed as the blob's `statRect` and `dragRect` fields, respectively.

If the blob is enabled, `SetProcRgnBlob()` draws the blob before returning.

The caller is responsible for disposing of the regions passed to `SetProcRgnBlob()`; those regions are used to construct new regions which are installed into the blob.

The drawing procedure should be defined like this:

```
pascal void
DrawProc (BlobHandle bDst, BlobHandle bSrc, Integer partCode)
{
}
```

bSrc is the blob that should be drawn and partCode indicates which region to draw. bDst indicates where to draw the given region of bSrc. This is to handle the case where a blob is glued onto another, since that means the drag region needs to be drawn, but in the location of the drag region of another blob. (If partCode is inStatBlob, bSrc and bDst are always the same.)

Since you may need to draw the static region at a location of a blob different than that of the blob to which the static region belongs, your draw procedure needs to be able to draw it at any arbitrary location.

```
pascal void
SetProcRectBlob (BlobHandle b, BDrawProcPtr proc,
```

`SetProcRectBlob()` installs `proc` as the blob's drawing procedure and installs rectangular regions into the blob corresponding to `dragFrame` and `statFrame`. The static region actually assigned is set to the difference of `statFrame` and `dragFrame`.

The bounding rectangles of the static and drag regions are installed as the blob's `statRect` and `dragRect` fields, respectively.

If the blob is enabled, `SetProcRectBlob()` draws the blob before returning.

The drawing procedure should be defined as described under `SetProcRgnBlob()` above.

4.6 Low-Level Blob Region Assignment

These routines are called by Blob Manager routines that also install pictures or procedures; they are not generally called by the host.

```
pascal void
SetBlobRgns (BlobHandle b, RgnHandle dragRgn, RgnHandle statRgn);
```

`SetBlobRgns()` installs regions into the blob corresponding to `dragRgn` and `statRgn`. The static region actually assigned is set to the difference of `statRgn` and `dragRgn`.

The caller is responsible for disposing of the regions passed to `SetBlobRgns()`; those regions are used to construct new regions which are installed into the blob.

```
pascal void
SetBlobRects (BlobHandle b, Rect *dragRect, Rect *statRect);
```

`SetBlobRects()` installs rectangular regions into the blob corresponding to `dragFrame` and `statFrame`. The static region actually assigned is set to the difference of `statFrame` and `dragFrame`.

```
pascal RgnHandle
BCalcRegion (BlobHandle b, short partCode);
```

`BCalcRegion()` returns a handle to a copy of the blob region corresponding to the given part

code. The caller is responsible for disposing of the region.

4.7 Graphic Operations on Blobs

In many ways, graphic operations on blobs are analogous to graphic operations on controls (`HideBlob()`, `HideControl()`; `ShowBlob()`, `ShowControl()`, etc.). There are some differences to be aware of. The most important is the difference in ownership of blobs and controls. When a control is created, the window it belongs to must be specified, and the control is only drawn in that window. A blob can be drawn in any window, but you must be sure that the current grafport is set correctly before performing any graphic blob operations.

pascal void
ShowBlob (BlobHandle b);

`ShowBlob()` makes the blob visible by enabling it and drawing it with `DrawBlob()`. If the blob is already enabled, `ShowBlob()` has no effect.

pascal void
ShowBlobSet (BlobSetHandle bSet);

Calls `ShowBlob()` for each blob in the set.

Note
`ShowBlob()` and `ShowBlobSet()` draw blobs immediately. Sometimes it is preferable to forestall drawing (for instance so that they will be drawn by the normal window updating procedure). To enable blobs without showing them, use `EnableBlob()` or `EnableBlobSet()`. These are useful in conjunction with invalidation of a window's `portRect` to trigger redrawing by the update procedure.

pascal void
HideBlob (BlobHandle b);

`HideBlob()` makes the blob invisible by disabling it and filling the bounding rectangles of the blob's static and drag regions with the background pattern of the current window's `grafPort`. The rectangles are added to the window's update region. If the blob is already disabled, `HideBlob()` has no effect.

pascal void
HideBlobSet (BlobSetHandle bSet);

Calls `HideBlob()` for each blob in the set.

Note
Since `HideBlob()` invalidates the bounds rectangles of the regions occupied by each blob, `HideBlobSet()` can result in a good deal of region calculation. For large blob sets the time involved in these calculations may be noticable. In such cases it is often much faster to invalidate the entire `portRect` of the `grafPort` first. If the blobs do not overlap other objects drawn in the window, the port can be validated after `HideBlobSet()` to forestall an update event.

pascal void
EnableBlob (BlobHandle b);

`EnableBlob()` sets the enabled bit in the blob's flag word. This does no redrawing.

pascal void
EnableBlobSet (BlobSetHandle bSet);

`EnableBlobSet()` enables every blob in the set. This does no redrawing.

pascal void
DisableBlob (BlobHandle b);

`DisableBlob()` clears the enabled bit in the blob's flag word. This does no redrawing.

pascal void
DisableBlobSet (BlobSetHandle bSet);

`DisableBlobSet()` enables every blob in the set. This does no redrawing.

pascal void
DrawBlob (BlobHandle b, short partCode);

If the blob is enabled, `DrawBlob()` draws the part of the blob indicated by `partCode`, using the blob's frame, picture or procedure, drawing modes and glue blob, in the manner discussed in Chapter 2 under "Blob Drawing." Drawing occurs even if the blob is already enabled. If `partCode` is `inFullBlob`, the entire blob is drawn.

pascal void
DrawBlobSet (BlobSetHandle bSet);

`DrawBlobSet()` calls `DrawBlob()` for each blob in the set with `inFullBlob` as the part code. This is commonly called in response to update events.

pascal void
DimBlobRgn (BlobHandle b, short partCode);

Dims the region of the given blob indicated by the part code. This should not be called for blobs that do dimming in some manner other than by dithering a gray pattern over the region.

pascal Boolean
BeginBlobDimDraw (BlobHandle b, short partCode);

Sets up to begin drawing in dimmed mode. Turns the pen gray if possible, otherwise does nothing. `BeginBlobDimDraw()` and `EndBlobDimDraw()` can be useful in procedure blobs that do their own dimming.

pascal void
EndBlobDimDraw (void);

Ends dim drawing. Restores the pen color to what it was before BeginBlobDimDraw() was called, if gray was available. Otherwise calls DimBlobRgn() to fill the affected blob region with gray.

pascal void
HiliteBlob (BlobHandle b, short partCode, short mode);

`HiliteBlob()` sets the drawing mode of the part of the blob indicated by `partCode`. If `partCode` is `inFullBlob`, the entire blob is set to the given mode. If `HiliteBlob()` changes the drawing mode of any of the parts of an enabled blob, they are redrawn appropriately.

pascal void

HiliteBlobSet (BlobSetHandle bSet, short partCode, short mode);

`HiliteBlobSet()` calls `HiliteBlob()` for every blob in the set, passing the given part code and drawing mode.

pascal void
GetBDimInfo (Pattern *p, short *mode);

Copies the current blob dimming pattern and drawing mode into the arguments.

pascal void
SetBDimInfo (Pattern p, short mode);

Sets the blob dimming pattern and drawing mode to the given pattern and mode. The defaults are `gray` and `patPic`, respectively.

pascal short
GetBDrawMode (BlobHandle b, short partCode);

`GetBDrawMode()` returns the drawing mode of the indicated part of the blob. `partCode` must be `inStatBlob` or `inDragBlob`.

pascal void
SetBDrawMode (BlobHandle b, short partCode, short mode);

`SetBDrawMode()` sets the drawing mode of the indicated part of the blob. The drawing mode of the entire blob is set if `partCode` is `inFullBlob`. `SetBDrawMode()` does no redrawing.

pascal short
GetFzBDrawMode (BlobHandle b, short partCode);

`GetFzBDrawMode()` returns the drawing mode of the indicated part of the blob, as it was before the blob was last frozen. This is used by routines that thaw frozen blobs, to determine how to restore their previous display state. `partCode` must be `inStatBlob` or `inDragBlob`.

4.8 Blob Transactions

4.8.1 High-Level Transaction Routines

The routines discussed below are used to perform automatic transaction handling. The descriptions are incomplete, as these routines are discussed more fully in the section “User-Initiated Transactions — the `lobClick()` Routine.”

pascal void
BlobClick (Point thePt, long t, BlobSetHandle dSet,

Call `BlobClick()` when the mouse is pressed to automatically perform blob transactions. The actions of `BlobClick()` are subject to the transaction permissions, the zoomback flags, the default dragging specifications and, if one is installed, the advisory filter function.

`dSet` may be `nil` if there is no donor set.

pascal short
BClickResult (void);

`BClickResult()` reports the result of the last call to `BlobClick()`. A value of zero means no transaction, otherwise the value indicates the transaction type.

```
typedef enum
{
                                                    noBcAct = 0,    /* nothin
                                                      bcGlue,       /* donor
                                                     bcClear,       /* glob u
                                                      bcXfer,       /* glob tr
                                                       bcDup,
                                                     bcSwap
};
```

If an advisory causes `BlobClick()` to abort, `BClickResult()` returns zero.

```
pascal void
BClickCast (BlobHandle *d1, BlobHandle *d2,
```

`BClickCast()` is used to find out which blobs were involved in a transaction (these are known collectively as the cast — as in cast of thousands — of the transaction). The meaning of the cast is dependent on the type of transaction performed. The values are undefined if `BClickResult()` returns zero.

Glue: **`db1` = donor glued, `rb1` = receptor glued to, `db2` = donor replaced by `db1` (`nil` if no replacement occurred), `rb2` = `nil`.**

Unglue: **`db1` = donor unglued, `rb1` = receptor unglued from, `db2`, `rb2` = `nil`.**

Transfer: **`db1` = donor transferred, `rb1` = source receptor, `rb2` = destination receptor, `db2` = donor replaced by `db1` (`nil` if no replacement occurred).**

Duplication: **same as transfer, except `db1` = donor duplicated.**

Swap: **`db1` = donor originally glued to `rb1`, `db2` = donor originally on `rb2`, `rb1` = source receptor, `rb2` = dest receptor.**

```
pascal void
SetBCPermissions (Boolean canUnglue, Boolean canXfer,
```

`SetBCPermissions()` sets the permission booleans that control the types of transactions `BlobClick()` is allowed to perform.

`canUnglue` is `true` if double-clicking detaches globs from receptors and if globs can be dragged off of receptors back to their donors. `canXfer` is `true` if globs can be transferred between receptors. `anDup` is `true` if globs can be duplicated onto other receptors. `canSwap` is `true` if globs can be swapped between receptors. `canRep` is `true` if transfers, swaps and duplications can cause a receptor's glob to be replaced by another. Note that if `canRep` is `false`, transfers and duplications can only be made to empty receptors, and swaps fail altogether.

Swaps take precedence over duplications, and both take precedence over transfers, if two or

more of these flags are on. All are subject to the value of the replacement flag, as noted above.

```
pascal void
GetBCPermissions (Boolean *canUnglue, Boolean *canXfer,
                                                    Boolean
```

`GetBCPermissions()` obtains the current permissions for the various types of transactions performed by `BlobClick()`.

```
pascal void
SetBCZoomFlags (Boolean uGlueZoom, Boolean bDragZoom);
```

`SetBCZoomFlags()` sets the flags determining whether `BlobClick()` does zooming on unglue transactions and bad drags. `uGlueZoom` determines whether `BlobClick()` will zoom an outline of the receptor drag region back to the donor when a donor is unglued from a receptor as a result of double-clicking the receptor. `bDragZoom` determines whether the outline of a dragged donor will be zoomed back where it was dragged from if it's not dragged somewhere it can be glued to.

```
pascal void
GetBCZoomFlags (Boolean *uGlueZoom, Boolean *bDragZoom);
```

`GetBCZoomFlags()` returns the current values of the flags controlling whether `BlobClick()` does zooming on unglue transactions and bad drags.

```
pascal void
SetBCAdvisory (BAdvisoryProcPtr p);
```

`SetBCAdvisory()` sets the address of the advisory filter function used by `BlobClick()`. If it's `nil`, `BlobClick()` does all the work itself. Otherwise, the function is consulted during transactions to see whether to continue processing or not. If the filter returns `false`, `BlobClick()` terminates early, and `BClickResult()` will return zero.

The advisory function should be defined like this:

```
pascal Boolean
Advisory (short mesg, BlobHandle blob)
{
}
```

The `mesg` argument indicates a click type or the transaction that `BlobClick()` proposes to perform. These messages are as follows:

```
typedef enum
{
                                        advDClick,      /* click i
                                        advRClick,      /* click i
                                          advGlue,      /* glue t
                                        advUnglue,      /* unglu
                                          advXfer,      /* transf
                                           advDup,      /* duplic
                                         advSwap        /* swap
};
```

The `blob` argument indicates the blob involved.

For more information, see section 3.7.2.

pascal void
GetBCAdvisory (BAdvisoryProcPtr *p);

`GetBCAdvisory()` returns the current address of the `BlobClick()` advisory filter function. A value of `nil` turns off any advisory currently in effect.

4.8.2 Low-Level Transaction Routines

The routines in this section can be used to perform blob transactions under program control. The transaction permissions and zoomback flags that are used to control `BlobClick()` have no effect on these routines.

pascal void
GlueGlob (BlobHandle d, BlobHandle r);

`GlueGlob()` glues the donor blob `d` to the receptor blob `r`. The donor's current glue count is incremented, and its drag region is dimmed if the count reaches or exceeds the donor's maximum glue count. If r already has a glue blob, it is unglued first (and undimmed if appropriate). The drag region of `r` is redrawn if it is enabled.

pascal void
ZGlueGlob (BlobHandle d, BlobHandle r);

`ZGlueGlob()` is the same as `GlueGlob()` except that an outline of the donor is zoomed the receptor for visual effect.

pascal void
UnglueGlob (BlobHandle b);

If `b` has a glue blob, `UnglueGlob()` unglues it. The glob's current glue count is decremented, and its drag region undimmed if the glue count goes below its maximum. The drag region of `b` is redrawn if it is enabled.

pascal void
ZUnglueGlob (BlobHandle b);

`ZUnglueGlob()` is the same as `UnglueGlob()` except that an outline of the glob is zoomed back from the receptor to the donor for visual effect.

pascal void
UnglueGlobSet (BlobSetHandle bSet);

For each blob in the given set, `UnglueGlobSet()` unglues its glue blob by calling `UnglueGlob()`.

pascal void
ZUnglueGlobSet (BlobSetHandle bSet);

For each blob in the given set, `ZUnglueGlobSet()` unglues its glue blob by calling `ZUnglueGlob()`.

pascal void

TransferGlob (BlobHandle r1, BlobHandle r2);

If `r1` has a glue blob, `TransferGlob()` unglues it from `r1` and glues it to `r2`. If `r1` is enabled, its drag region is redrawn. Any glob already glued to `r2` is unglued first (and undimmed if appropriate). The drag region of `r2` is redrawn if it is enabled.

pascal void
DupGlob (BlobHandle r1, BlobHandle r2);

pascal void
ZDupGlob (BlobHandle r1, BlobHandle r2);

If `r1` has a glob, `DupGlob()` glues the glob to `r2` as well. Any glob already glued to `r2` is unglued first (and undimmed if appropriate). The drag region of `r2` is redrawn if it is enabled. `ZDupGlob()` is the same but zooms an outline of the glob from `r1` to `r2`.

pascal void
SwapGlob (BlobHandle r1, BlobHandle r2);

If `r1` and `r2` have globs, `SwapGlob()` glues the glob attached to `r1`onto `r2`, and glues the glob attached to `r2` onto `r1`. If either `r1` or `r2` is enabled, its drag region is redrawn.

pascal void
IncBlobGlue (BlobHandle b);

`IncBlobGlue()` increments the current glue count of the blob, and dims it if it is enabled and the count reaches the blob's maximum glue count.

pascal void
DecBlobGlue (BlobHandle b);

`DecBlobGlue()` decrements the current glue count of the blob, and undims it if it is enabled and the count goes below the blob's maximum glue count.

4.9 Blob Movement

The following routines may be used to move blobs or parts of blobs to a relative or absolute location.

Note
Moving one region of a blob without moving the other region may give undesirable results; you could end up with overlapping regions.

pascal void
OffsetBlob (BlobHandle b, short partCode,

`OffsetBlob()` offsets the indicated part of the blob a distance of `dh` horizontally and `dv` vertically. If `partCode` is `inFullBlob`, the entire blob is offset as a unit. If the blob is enabled, it is hidden and redrawn at the new location.

pascal void
MoveBlob (BlobHandle b, short partCode,

`MoveBlob()` moves the indicated part of the blob so that its top left corner is at the horizontal

and vertical coordinates `h` and `v`. If `partCode` is `inFullBlob`, the entire blob is moved as a unit; the top left corner of the static region bounding rectangle is placed at `(h, v)`, while the drag region remains at the same location relative to the static region as it was prior to the call to `MoveBlob()`. If the blob is enabled, it is hidden and redrawn at the new location.

4.10 Blob Hit Testing

pascal short
TestBlob (BlobHandle b, Point thePoint);

`TestBlob()` returns a part code (`inStatBlob` or `inDragBlob`) if `thePoint` lies within the blob's static or drag region, zero otherwise.

Only undimmed regions of active blobs are tested.

pascal short
FindBlob (Point thePoint, BlobSetHandle bSet, BlobHandle *bhPtr);

If `thePoint` does not lie within any blob in the given blob set, `FindBlob()` returns `zero`. If `thePoint` lies within a blob in the given blob set, `FindBlob()` returns a handle to it in `b`, and the function result is the part code of the part of the blob that the point is in (either `inStatBlob` or `inDragBlob`).

Only undimmed regions of active blobs are tested.

4.11 Blob Tracking and Dragging

pascal void
DragBlob (BlobHandle b, Point startPoint,
Rect *lim

When the mouse button is pressed in a blob, call `DragBlob()` with the point at which the mouse was pressed. `ragBlob()` follows movements of the mouse by dragging around a gray outline of the blob until the mouse button is released. If the mouse button is released inside of the `slopRect`, the blob is erased and redrawn at the new location.

pascal long
TrackBlob (BlobHandle b, short partCode, Point startPoint,

`TrackBlob()` drags around a gray outline of a the indicated blob part (the entire blob if `partCode` is `inFullBlob`), until the mouse button is released. If the button is released outside of the `slopRect`, `TrackBlob()` returns `0x80008000`. Otherwise the vertical and horizontal differences between the final mouse point and the `startPoint` are returned in the high and low order words of the result.

pascal void
DDragBlob (BlobHandle b, Point startPoint);

pascal long
DTrackBlob (BlobHandle b, short partCode, startPoint);

`DDragBlob()()` and `DTrackBlob()()` are the same as `DragBlob()` and `TrackBlob()`, respectively, except that the Blob Manager's default `limitRect`, `slopRect` and drag axis are used.

pascal void
SetBDragRects (Rect *limitRect, Rect *slopRect);

`SetBDragRects()` sets the Blob Manager's default limit and slop drag rectangles to `limitRect` and `slopRect`, respectively.

pascal void
GetBDragRects (Rect *limitRect, Rect *slopRect);

`GetBDragRects()` returns the Blob Manager's default limit and slop drag rectangles in the `limitRect` and `slopRect` parameters, respectively.

pascal void
SetBDragAxis (short axis);

`SetBDragAxis()` sets the Blob Manager's default drag axis to the given value.

pascal short
GetBDragAxis (void);

`GetBDragAxis()` returns the Blob Manager's default drag axis value.

pascal Boolean
BTrackMouse (BlobHandle b, Point startPt, short partCode);

The blob tracking routines track the blob by making it follow mouse movements. `BTrackMouse()` tracks the mouse to see whether it remains inside of a stationary blob. When the mouse is clicked in a blob, pass a handle to the blob, the mouse position and a part code to `BTrackMouse()`. `BTrackMouse()` highlights the part whenever the mouse is in it, and returns true if the mouse is released in the same part. Ths routine is useful to applications that wish to use blobs that act in a manner similar to controls such as push buttons.

4.12 Blob Match Testing and Match Result Display

pascal Boolean
InBlobMatchSet (BlobHandle b1, BlobHandle b2);

`InBlobMatchSet()` returns `true` if `b1` is found in the match set of blob `b2`, `false` otherwise.

pascal Boolean
BlobQuiet (BlobHandle b);

If the blob is quiet according to the rules for blob matching, `lobQuiet()` returns `true`, otherwise `false`. Blob matching is discussed in more detail under "Overview of Blob Behavior."

pascal void
SetBQuietTest (BQuietProcPtr proc);

Installs a routine to be called to test whether a blob is quiet or not. If `proc` is `nil`, the default routine is used. Otherwise the routine should be declared like this:

```
pascal Boolean
QuietTest (BlobHandle b)
```

```
{
}
```

The argument indicates the blob to test. The routine should return `true` or `false` depending on whether or not the blob should be considered quiet.

```
pascal Boolean
BlobSetQuiet (BlobSetHandle bSet);
```

If every blob in the set is quiet according to the rules for blob matching, `BlobSetQuiet()` returns `true`, otherwise `false`. Blob matching is discussed in more detail under "Overview of Blob Behavior."

```
pascal void
FreezeBlob (BlobHandle b);
```

`FreezeBlob()` freezes the blob. This causes its current state, including its drawing modes and whether it is enabled or not, to be saved. Freezing a frozen blob has no effect.

```
pascal void
FreezeBlobSet (BlobSetHandle bSet);
```

`FreezeBlobSet()` freezes the enabled blobs in the set.

```
pascal void
ThawBlob (BlobHandle b);
```

If the blob is frozen, `ThawBlob()` unfreezes it and restores it to its pre-freeze state.

```
pascal void
ThawBlobSet (BlobSetHandle bSet);
```

`ThawBlobSet()` thaws the enabled blobs in the set.

```
pascal void
BlobFeedback (BlobSetHandle bSet,
```

For each enabled blob in the set, `BlobFeedback()` freezes it and calls `BlobQuiet()`. If the blob is quiet, it is drawn in `quietMode` drawing mode, otherwise it is drawn with `noisyMode` drawing mode. Since `BlobFeedback()` freezes enabled blobs, `ThawBlobSet()` must be called to restore them.

4.13 Blob Condition Testing

```
pascal void
SetBlobFlags (BlobHandle b, short bitMask);
```

For each bit that is set in the bit mask, `SetBlobFlags()` sets the corresponding bit in the flag word of the blob.

```
pascal void
ClearBlobFlags (BlobHandle b, short bitMask);
```

For each bit that is set in the bit mask, `ClearBlobFlags()` clears the corresponding bit in the

flag word of the blob.

pascal short
TestBlobFlags (BlobHandle b, short bitMask);

For each bit that is set in the bit mask, `TestBlobFlags()` returns a one or a zero in the corresponding bit of the result, depending on whether the bit is set or cleared in the blob's flag word.

pascal Boolean
BlobEnabled (BlobHandle b);

`BlobEnabled()` returns `true` if the enable bit of the blob's flag word is set, `false` otherwise.

pascal Boolean
BlobDimmed (BlobHandle b, short partCode);

`BlobDimmed()` returns `true` if the given region of the blob is dimmed, `false` otherwise. If `partCode` is `inFullBlob`, the drag region and the static region must both be dimmed for `BlobDimmed()` to return `true`.

pascal Boolean
BlobFrozen (BlobHandle b);

`BlobFrozen()` returns `true` if the frozen bit of the blob's flag word is set, `false` otherwise.

pascal Boolean
BlobActive (BlobHandle b);

`BlobActive()` returns `true` if the blob is active, false otherwise. A blob is active if it is enabled but not frozen.

pascal Boolean
CanGlue (BlobHandle b);

`CanGlue()` returns `true` if `b` is not currently glued to the number of receptors specified by its `glueMax` field, *i.e.*, if it can be glued onto another receptor.

pascal Boolean
PicBlob (BlobHandle b);

`PicBlob()` returns `true` if `b` is a picture blob, `false` if it's not (i.e., if it's a procedure blob).

4.14 Miscellaneous Routines

pascal BlobHandle
GetBlobHandle (BlobSetHandle bSet, short i);

`GetBlobHandle()` returns the handle of the *i*-th blob in the given blob set, if *i* is in the range from `0` (the first blob in the set) to the number of blobs in the set – 1. `GetBlobHandle()` returns `nil` otherwise.

This function allows blob sets to be treated like one-dimensional arrays, albeit in a rather inefficient manner.

pascal short
GetBlobIndex (BlobSetHandle bSet, BlobHandle b);

`GetBlobIndex()` returns the sequential position of blob `b` within the blob set `bSet`. The first blob has index `0`.

pascal short
BlobSetSize (BlobSetHandle bSet);

`BlobSetSize()` returns the number of blobs in the set.

pascal short
BlobRand (short max);

`BlobRand()` returns a random integer between `0` and `max-1`, inclusive.

pascal void
SetBlobRand (BRandProcPtr f);

If the default properties of `BlobRand()` are unsatisfactory, they may be changed by installing a different generator. The generator function should be declared like this:

```
pascal short
MyRand (short max)
{
}
```

The generator should return an integer between zero and `max`, inclusive. It is installed by passing its address to `SetBlobRand()`.

If `f` is `nil`, the default generator is reinstalled.

pascal void
ShuffleBlobSet (BlobSetHandle bSet);

Randomizes the locations of the blobs in the given set. The locations of the blobs in the set remains the same, but assignment of individual blobs to those locations changes. If the blobs are enabled there will be lots of movement on the screen, so you should call `ideBlobSet()` before calling `ShuffleBlobSet()` and `ShowBlobSet()` afterward, if you wish to avoid this.

Note
When blobs are shuffled they are moved. If they are also enabled, moving them causes them to be hidden and redrawn (`HideBlob()`/`ShowBlob()`), so the same remarks about region calculations made under the description of `HideBlobSet()` apply to `ShuffleBlobSet()`. It is sometimes preferable to invalidate the window `portRect` before shuffling and validate it afterwards, to speed up the shuffle and to avoid an update event.

Warning
`ShuffleBlobSet()` may not work correctly if any of the blobs in the set have empty drag regions. This does not often occur.

pascal void
ShuffleGlobSet (BlobSetHandle bSet);

`ShuffleGlobSet()` randomly swaps around the globs glued to the blobs in `bSet`. Unlike

`ShuffleBlobSet()`, no region invalidation is involved.

```
pascal void
BMgrZoomRect (Rect *r1, Rect *r2);
```

`BMgrZoomRect()` zooms a gray rectangle from the location of `r1` to the location of `r2`. At the beginning of the zoom, the gray rectangle is the shape of `r1`. During the zoom the rectangle changes shape so that at the end of the zoom it is the shape of `r2`. `BMgrZoomRect()` is nondestructive to the display.

```
pascal void
BlobLoopProc1 (BLoopProcPtr1 p, BlobSetHandle bSet);

pascal void
BlobLoopProc2 (BLoopProcPtr2 p, BlobSetHandle bSet,
```

`BlobLoopProc1()` and `BlobLoopProc2()` are used to implement operations on sets of blobs. For each blob in the set, `BlobLoopProc1()` passes the blob to the given procedure. For instance, `ShowBlobSet()` is implemented as

```
BlobLoopProc (ShowBlob, bSet)
```

The function called by `BlobLoopProc1()` should be defined like this:

```
pascal void
Func (BlobHandle b)
{
}
```

`BlobLoopProc2()` is similar, except that it also passes a part code. For instance, `DrawBlobSet()` is implemented as

```
BlobLoopProc2 (DrawBlob, bSet, inFullBlob)
```

The function called by `BlobLoopProc2()` should be defined like this:

```
pascal void
Func (BlobHandle b, short partCode)
{
}
```

Although the third argument is used as a part code by the Blob Manager, in fact `BlobLoopProc2()` can be used to pass a blob and any `short` argument.