3. Using the Blob Manager

This section describes how to write a blob scenario using the Blob Manager.

3.1 General Overview

The header file *BlobMgr.h* should be included in any source file using Blob Manager calls, in order to access Blob Manager data types, constants and other definitions. Your application project document must include the*BlobMgr* project or library document so that Blob Manager functions can be linked in.

For each blob set that is to be used, you call `NewBlobSet()`, which returns a handle to an empty set. Call `NewBlob()` once for each blob that is to be added to the set, passing the blob set handle to `NewBlob()` so that it knows which set to attach the blob to. Then the blob image is constructed, either by creating a picture or installing an image-drawing procedure.

If the scenario is not such as to require special treatment, it is often sufficient to set the `BlobClick()` flags, then call `BlobClick()` whenever there is a mouse click in a window containing blobs. After each click you can check which receptors have globs to see whether a termination state has been reached.

Before exiting, call `DisposeBlobSets()` to shut down the Blob Manager.

The Blob Manager Demo source code may be examined as an example of an application that sets up several scenarios with widely differing requirements.

3.2 Blob Manager Data Types

Blobs are organized into *blob sets*. Every blob in the set is represented internally by a *blob record* containing all the pertinent information about the blob. The blob record contains the following types of information.

- **A flags word. This indicates whether the blob is enabled or frozen, the drawing modes of the static and drag regions, and other sorts of information.**
- **A flags word used by routines that freeze and thaw blobs. This field is for internal use only; user programs shouldn’t mess with it.**
- **A handle to the picture or the procedure that draws the blob.**
- **Handles to the static and drag regions.**
- **The bounds rectangles of the static and drag regions at the time the blob image was assigned to the blob. This is for internal use only.**
- **A count of the number of times that the blob may be glued onto other blobs before its drag region goes dim and becomes unavailable for dragging. This is used for blobs serving as donors.**
- **A count of the number of blobs the blob is actually glued to. This is used for blobs serving as donors.**
- **A handle to the set of blobs that match the blob. This is used for blobs serving as receptors.**
- **A handle to the current glob. This is used for blobs serving as receptors.**

- **A reference value that is reserved for use by your application. You specify an initial reference value when the blob is created, and can read or change the reference value whenever desired. You might use it to store a blob ID number, or a handle to a data structure containing other information about the blob that is used to perform operations the Blob Manager does not provide.**
- **A handle to the next blob in the set.**

The blob record is defined as follows:

```
typedef struct BlobRecord                    BlobRecord, *BlobPtr, **BlobHandle;

struct BlobRecord
{
                                                                    Integer
                                                                    Integer
                                                                      union
                                                                          {
                                                                              PicHand
/* blob picture (and original frame) */
                                                                              BDrawP
/* blob drawing procedure */
                                                                    } bPicProc;
                                                                    RgnHandle
                                                                    RgnHandle
                                                                       Rect
                                                                       Rect
                                                                    Integer
                                                                    Integer
                                                                 MatchHandle
                                                                  BlobHandle
                                                                       long
                                                                  BlobHandle
};
```

You can store into and access most of a blob record's fields with Blob Manager routines, so normally you don't have to know the exact field names.

A blob record may contain a handle to a list of blobs considered to be matches. Initially this handle is `nil`. If it's non-`nil`, each element in the match list has the following structure:

```
typedef struct MatchRecord                   MatchRecord, *MatchPtr, **MatchHandle;

struct MatchRecord
{
                                                      BlobHandle    mBlob;
                                                     MatchHandle    nextMat
};
```

Every blob set has a header containing handles to the first and last blob in the set:

```
typedef struct BlobSetRecord              BlobSetRecord, *BlobSetPtr, **BlobSetHandle;

struct BlobSetRecord
{
```

```
                                        BlobHandle      firstBlob
                                        BlobHandle      lastBlob
                                    BlobSetHandle       nextBlob
};
```

The `nextBlobSet` field of the `BlobSetRecord` is used internally by the Blob Manager to maintain a master list of all blob sets created and should not be changed by user programs.

All blob sets can be disposed of at program termination time by calling `DisposeBlobSets()`. To dispose of an individual blob or blob set, use `ClobberBlob()` or `ClobberBlobSet()`.

Note
Although sets as mathematical entities are unordered, blob sets in particular have an ordering induced on them by the implementation (sets are treated as lists). When blobs are created, they are added to the end of their set, thus the i-th blob in a set — a concept with no mathematical meaning — is easily obtained with `GetBlobHandle()`. This allows treatment of blob sets as single- dimensional arrays. More complex data structures can be induced on a blob set by making use of this fact.

3.3 Creating a Blob Set

To create a new blob set, declare a `BlobSetHandle` variable and call `NewBlobSet()`:

```
BlobSetHandle                                   myBlobSet;

myBlobSet = NewBlobSet ();
```

`NewBlobSet()` allocates a new `BlobSet` record and returns a handle to it, or `nil` if there is insufficient memory.

Once you have the `BlobSet` handle, you can call `NewBlob()` as many times as necessary to create the blobs that should make up the set:

```
BlobHandle                                      b;

b = NewBlob (&myBlobSet, enable, maxGlue, mustMatch, refCon);
```

`NewBlob()` takes a pointer to the `BlobSet` handle, and parameters indicating whether the blob is initially enabled, the maximum glue count, whether or not the blob needs an explicit match, and a reference constant. `NewBlob()` allocates a new blob record, initializes it, adds it to the given blob set, and returns a handle to it. If there is insufficient memory to allocate the blob, `NewBlob()` returns `nil`.

The `enable` parameter is `true` or `false`, and determines whether or not the blob will be drawn as soon as its image picture or drawing procedure is installed. `maxGlue` indicates the number of other blobs the drag region can be glued to before the blob goes dim. `mustMatch` is `true` or `false` and indicates whether or not the blob requires an explicit match to be quiet. `refCon` is a reference constant and can be used for whatever purpose you want.

A newly-created blob cannot be used for anything; it is given empty static and drag regions, and it has no image. Nor can the blob be drawn until you inform the Blob Manager how to draw its image.

The appearance, or image, of a blob is arbitrary, subject to the constraint that drawing is always done inside of the static and drag regions. The image is arbitrary because it can consist of anything you can draw with QuickDraw. The constraint is enforced by clipping the picture to the appropriate region when it is drawn.

There are two methods for defining blob images. The first is to issue a set of QuickDraw calls as a picture definition, and install the picture into the blob record. Blobs imaged this way are *picture blobs*. Whenever the Blob Manager draws a picture blob, it replays the picture, clipped to the region of the blob that is to be drawn. This effectively draws only that part of the picture that lies within the desired region.

When the drag region of a blob A is glued to another blob B, the drag region of A is redrawn in a similar manner, except that the picture is clipped to the drag region of B. The picture for A is mapped to the picture frame of B's drag region so that the picture for A is resized to fit B.

When a picture blob is dimmed, it's drawn the usual way by replaying the picture. Then a gray pattern is drawn over it in `patBic` mode to erase some of it.

The second method of defining a blob image is to have the blob passed to a procedure that knows how to draw it. Blobs imaged this way are *procedure blobs*.

There are trade-offs to consider when deciding which blob imaging method should be used.

Advantages of picture blobs:

- **The picture method is often easier because the QuickDraw picture mechanism allows pictures to be replayed in any location or size.**

Disadvantage of picture blobs:

- **For sets of blobs whose members all look very similar, memory can be wasted since a picture must be installed in each blob (pictures are not shared between blobs).**
- **Some QuickDraw calls, such as `CopyMask()` and `PlotCIcon()`, are not recorded in pictures. In such cases, you must use procedure blobs.**

Advantages of procedure blobs:

- **A drawing procedure can be shared among several blobs, which can result in a lower memory requirement than generating and storing pictures for each blob.**
- **Blob appearance may be modified under program control without creating a new picture.**
- **The procedure can be written to draw only the region of the blob indicated by partCode. This can be more efficient than drawing the entire blob (which is essentially what happens for picture blobs). If you want, you can still draw the entire blob for simplicity, because the clipping is still set so that drawing only appears in the appropriate region.**
- **Dimmed procedure blobs can be drawn to look nicer. On monitors capable of it, the procedure can draw in true gray, which looks better than splatting gray over the region after drawing it normally. The drawing procedure can either let the Blob Manager set the pen color before the procedure is called and restore it afterward, or the procedure can do its own dimming.**

Disadvantages of procedure blobs:

- **The procedure must know how to draw the blob in any location, and, if blobs are of different sizes, how to adjust for size differences.**
- **Drawing procedures must be careful to avoid certain QuickDraw calls such as `PenNormal()`, since if the pen has been changed to gray, that undoes the pen color change.**

  Note

It is possible to mix picture and procedure blobs within a scenario. In particular, picture blobs may be glued onto procedure blobs, and vice versa.

3.3.1 Picture Blob Creation

The process of blob picture creation is analogous to the process of picture creation in general: you open the blob, draw its image, and close it. In fact, the routines that open and close blobs use the picture creation routines `OpenPicture()` and `ClosePicture()`, so you should be familiar with the operation of those routines.

To set up a picture blob, call `OpenBlob()`, draw the image that determines what the blob should look like, then call either `CloseRgnBlob()` or `CloseRectBlob()`:

```
OpenBlob ();
/* draw image */
CloseRgnBlob (b, dragRgn, statRgn);

OpenBlob ();
/* draw image */
CloseRectBlob (b, dragRect, statRect);
```

`OpenBlob()` calls `OpenPicture()`, which hides the pen, so nothing appears on the screen while the picture is being formed. The blob-closing functions take a handle to the blob that the picture should be assigned to, and the regions or rectangles that describe the drag and static regions. `CloseRgnBlob()` is more general, but `CloseRectBlob()` can be more convenient if the outlines of the blob regions can be described as rectangles.

`CloseRectBlob()` and `CloseRgnBlob()` partition the general blob region into the static and drag regions, and assign the picture to the blob. Some preprocessing is done to the picture to avoid the QuickDraw picture replay quirk that sometimes results in nothing being drawn if the picture is redrawn into a different sized frame.

The regions or rectangles you specify don’t need to be mutually exclusive: the drag region takes precedence, so that any overlap between the static and drag regions is assigned to the drag region.

Here’s an example that creates a blob consisting of two adjacent circles:

show

Note that this code disposes of the regions after passing them to `CloseRgnBlob()`; this is done because `CloseRgnBlob()` installs *copies* of the regions into the blob.

Here’s an example of a blob consisting of one rectangle within the other, where the inner rectangle is the drag region:

show

Any picture blob which is enabled when it is closed is drawn by the close procedure as soon as the picture is installed.

If you redefine a blob’s image, the old picture (if any) is disposed of.

Glob images of picture blobs are scaled to the receptor blob drag region. If the glob’s drag region is a different size or shape than that of the receptor, scaling is done automatically by the QuickDraw picture drawing mechanism.

*** Discuss clipping region problems here, and how the Blob Manager takes care of them. ***

3.3.2 Procedure Blob Creation

To set up a procedure blob, define the regions or rectangles describing the static and drag regions, then call `SetProcRgnBlob()` or `SetProcRectBlob()`. These functions take a blob handle, a pointer to the procedure that draws the blob, and the regions or rectangles:

```
SetProcRgnBlob (b, DrawProc, dragRgn, statRgn);

SetProcRectBlob (b, DrawProc, dragRect, statRect);
```

To set up the picture blobs described above as procedure blobs instead, you could do this:

```
show
```

Note that this code disposes of the regions after passing them to `SetProcRgnBlob()`; this is done because `SetProcRgnBlob()` installs *copies* of the regions into the blob.

When a procedure blob needs to be drawn, the Blob Manager calls the drawing function, which should look 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.

With these considerations in mind, the drawing procedures `DrawCircleBlob()` and `DrawRectBlob()` can be written like this:

```
pascal void
DrawCircleBlob (BlobHandle bDst, BlobHandle bSrc, Integer partCode)
{
Rect                                                                    r;

                                                          r = BDragBox (bDst);
                                                               EraseOval (&r);
                                                              FrameOval (&r);
                                                         r = BStatBox (bDst);
                                                               EraseOval (&r);
                                                              FrameOval (&r);
}

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

```
    Rect                                                    r;

                                                        r = BDragBox (bDst);
                                                          EraseRect (&r);
                                                          FrameRect (&r);
                                                      r = BStatBox (bDst);
                                                          EraseRect (&r);
                                                          FrameRect (&r);
    }
```

If the glob is a procedure blob, scaling is done only if the drawing procedure is intelligent enough to do so.

The `bRefcon` field often used to distinguish blobs. If the source and dest blobs are different, that means draw a glob on a blob.

3.4 Blob Drawing

If the blob is not enabled or has no image (i.e., no picture, no drawing procedure), the Blob Manager won't draw it. Period. If the blob is enabled, the Blob Manager can be told to draw either or both the static and drag regions. The image drawn in the static region is always the blob's own image, clipped to the static region. The image drawn in the drag region is either its own (if it has no glob), or its glob's drag image (if it has a glob), in either case clipped to its own drag region.

A blob region is drawn either normally or dimmed, depending on its drawing mode:

```
    typedef enum
    {
                                                    normalDraw = 1,     /* draw b
                                                      dimDraw = 2
    };
```

Each region has its own drawing mode, so each region can be drawn normally or dimmed independently of the other.

By default, a picture blob region is dimmed by drawing it normally, then filling the region with `gray` in `patBic` mode. (If the image is initially drawn this way, dimming it will make it invisible, so be careful.) The default dimming behavior may be changed with `SetBDimInfo()`, which takes a QuickDraw drawing mode and a pattern. For instance, a host can make dimmed blobs disappear by changing the dimming pattern to `black`, or make dimmed blobs appear inverted relative to their normal appearance by changing the pattern to `black` and the mode to `patXor`. The current dimming pattern and drawing mode may be obtained with `GetBDimInfo()`.

Procedure blobs are dimmed by changing the pen to gray during drawing if the system and monitor support gray. If a gray pen is not available, dimming is done by filling the region after drawing the same way as for picture blobs. Procedure blobs also have the option of telling the Blob Manager not to do the dimming, but to let the drawing procedure do it.

Blobs are made visible or invisible by calling `ShowBlob()` or `HideBlob()`, respectively. Enabled blobs may be redrawn (e.g., in response to update events) by calling `DrawBlob()`, which draws the blob with the current drawing modes. Entire sets may be hidden, drawn or redrawn with `HideBlobSet()`, `ShowBlobSet()` or `DrawBlobSet()`.

3.5 Blob Hit-testing, Tracking, Dragging and Moving

For many scenarios, the `BlobClick()` routine is entirely sufficient to handle all transactions. Should the

host itself wish to determine which blobs mouse clicks fall in, or do tracking or dragging operations itself, it can do so.

The `TestBlob()` routine determines whether a mouse point is in a given blob or not and returns a *part code* to indicate whether the point is in the static region or the drag region, or zero to indicate that the point is not in the blob.

```
typedef enum
{
                                                    inStatBlob = 1,     /* static
                                                   inDragBlob = 2,      /* drag r
                                                     inFullBlob = 3     /* entire
};
```

A blob must be active (enabled, not frozen), and the part of the blob that the mouse is in must be undimmed, for the blob to be considered hit.

`FindBlob()` may be used to determine which, if any, of a set of blobs a mouse point lies in.

When the mouse button is pressed in a blob, the Blob Manager may be called to drag around a gray outline of the blob or part of the blob. The blob may be dragged to the point where the mouse is released, or the Blob Manager may simply return the result of the blob drag. `TrackBlob()` drags an outline of the tracked part around and returns the result of the drag. `DragBlob()` tracks the part and then moves it to the endpoint of the drag.

Blob dragging involves the concepts of the limit and slop rectangles, and of the drag axis. These are discussed under the `DragGrayRgn()` routine in the Window Manager manual. The routines `DTrackBlob()` and `DDragBlob()` allow tracking and dragging with the Blob Manager's default drag axis and rectangles. These are set when the Blob Manager is initialized. The default limit and slop rectangles are the size of the standard Macintosh screen and a wide-open rectangle, respectively, but the limit rectangle should usually be set to the `portRect` of the window in which the scenario lives.

Blobs may also be offset or moved by the host program with `OffsetBlob()` or `MoveBlob()`. When a blob's picture is created, the static and drag regions are mutually exclusive. If both regions of a blob are moved together, they move in synchrony, so that the regions remain non- overlapping. If either region is moved independently of the other, the appearance of the blob may be affected adversely, especially if they overlap.

The blob tracking routines track the blob by making an outline of it follow the movements of the mouse. In constrast, `BTrackMouse()` is used to track whether the mouse remains in a stationary blob when the mouse moves. This is analogous to `TrackControl()`. It takes a blob, a starting point and a part code. The region specified by the part code is highlighted (by inverting it) whenever the mouse is inside it. If the mouse is released inside the region, `BTrackMouse()` returns true. Conceptually, this operation is identical to the way push buttons are tracked when clicked with the mouse. Since the appearance of blobs is less constrained, one can easily implement “controls” of highly arbitrary appearance.

3.6 Blob Manager State Information and Multiple Blob Scenarios

Applications implementing multiple scenarios should take note of the following information. Certain Blob Manager operations use parameters that may be set by the host application to implement particular scenarios. If an application supports more than one scenario, and the parameter settings differ between them, it is important to set them when switching from one scenario to another. Typically each scenario gets its own window, so you switch scenarios when an activate event occurs.

Settable parameters include the dimming pattern and drawing mode, the default limit and slop rectangles

and the default drag axis, and the `BlobClick()` zoomback and transaction permissions flags, and the advisory function. (The advisory is described in the section discussing miscellaneous topics.)

The dimming pattern and drawing mode are used whenever blobs are drawn. The default rectangles and drag axis are used by the default tracking and dragging routines. The default tracking routine, the flags, and the advisory are all used in turn by `BlobClick()`. This means that any scenario using `BlobClick()` is subject to the current settings of all these parameters.

3.7 Miscellaneous Topics

3.7.1 Secondary Data Structures

When blobs are created, they are added to the end of their set. Thus they form an implicit one-dimensional array, the *i*-th member of which can be accessed as `GetBlobHandle (bSet, i)`. For small blob sets, this may be perfectly adequate. For sets of larger size, `GetBlobHandle()` may be too inefficient as a general access method. In such cases, it is beneficial to induce secondary structures on the blob set.

An explicit array can be used to speed up access:

```
BlobSetHandle                                                    bSet;
BlobHandle                                                   b[maxBlob];
Integer                                                              i;

bSet = NewBlobSet ();
for (i = 0; i < maxBlob; ++i)
{
                                                    b[i] = NewBlob (bSet, ... );
                                                  /* create blob image here */
}
```

Higher-order structures may easily be created in analogous fashion. For instance, creation of blobs to populate a checkboard may be done as follows:

```
BlobSetHandle                                                    bSet;
BlobHandle                                                 board[8][8];
Integer                                                              i;

bSet = NewBlobSet ();
for (i = 0; i < 8; ++i)
{
                                                      for (j = 0; j < 8; ++j)
                                                                      {
                                                                          b[i][j] = N
                                                                          /* create
                                                                      }
}
```

In both examples, blobs may be accessed directly in constant time through array references, rather than by looking through a list. Reference the the checkerboard blobs are much quicker; using `GetBlobHandle()`, 64/2 = 32 blobs must be scanned before finding the right one, on average.

3.7.2 `BlobClick()` Revisited: The Advisory Filter

In simple Blob Manager applications, the host creates the blobs to be operated on, sets the transaction

permissions and then passes mouse clicks to `BlobClick()`, which carries out its job without assistance from the host. When it returns, a transaction has either been completed or not, and the host may examine the blob sets to see whether a termination state has been reached.

Often illegal moves can be ruled out by freezing certain blobs or by setting the transaction permissions appropriately. For instance, in tic-tac-toe, all illegal moves can be prevented before the user clicks the mouse simply by freezing every square on the board that is already occupied. The same end can be achieved here by turning off replace permission. Not every situation is so simple, however, and in such cases it is desirable to know something about transactions while they are actually occurring.

For example, in a game of checkers, illegal moves cannot be prevented by freezing certain board positions, because the set of legal moves is dependent on the piece the user wishes to move. If it were known which piece the mouse was clicked in and which square of the board it was dragged to, the legality of the move could be assessed and the move either allowed or disallowed before the piece was actually moved. In a game of solitaire, only certain cards may legally be laid on each other card, but again the set of cards that may be placed on each card varies. Knowing which cards the user planned to move, and where, is information that could be used to disallow transactions which might be legal according to the general transaction permissions, but illegal according to the semantics of the game.

`BlobClick()` allows the host to specify an advisory filter function to cope with scenarios that may be too complex to handle with the normal mechanism. Installation of an advisory function allows the host access to information about transactions in progress, so that it can, in effect, “call the shots” on transactions without having to perform the mechanics of hit-testing, dragging, etc. It should be declared to take two arguments.

```
pascal Boolean
BCFilter (short message, BlobHandle blob)
{
}
```

`BlobClick()` calls this filter in several different circumstances. The `message` argument indicates a click type or the transaction that `BlobClick()` proposes to perform. These messages are as follows:

```
typedef enum
{
                                                    advDClick,      /* click in
                                                    advRClick,      /* click in
                                                      advGlue,      /* glue tr
                                                    advUnglue,      /* unglue
                                                      advXfer,      /* transf
                                                      advDup,       /* duplic
                                                     advSwap        /* swap
};
```

The `blob` argument indicates the blob involved.

(i) When a mouse click falls in an active donor blob, the message is `advDClick` and the blob argument is the donor in which the mouse was clicked.

(ii) When the donor from (i) is dragged onto an active receptor, the message is `advGlue` and the blob argument is the receptor onto which the donor was dragged.

(iii) When a mouse click falls in an active receptor with a glob attached to it, the message is `advRClick` and the blob argument is the receptor in which the mouse was clicked.

(iv) When the glob from (iii) is dragged onto a different active receptor, the blob argument is the receptor the glob is dragged onto, and the message is `advXfer`, `advDup` or `advSwap`, depending on the transaction that `BlobClick()` proposes to perform. If the glob is dragged

back to the donor it came from, the message is `advUnglue`, and the blob argument is the receptor the glob was dragged from.

(v) When an active receptor with a glob is double-clicked in its drag region, the message if `advUnglue` and the blob argument is the receptor in which the mouse was double-clicked.

Note that the advisory function is called twice for most transactions. It will not be called at all if the mouse click doesn't fall in something draggable, and will only be called once if a donor or glob is hit but not dragged somewhere it can be glued.

The advisory function does not extend the scope of allowable transactions or override the general transaction permissions. It is not even called unless `BlobClick()` considers the action done by the user to be legal according to those permissions, so the advisory can only further restrict the user's actions. Thus if a donor is dragged to a receptor already having a glob, but replace permission is turned off, the advisory is called only when the donor is clicked, not when the donor is dragged to the receptor (`BlobClick()` wouldn't consider replacing the glob that is already there, anyway).

The advisory function examines its arguments, takes whatever action it deems appropriate, then returns a boolean value to `BlobClick()` indicating whether to continue processing or not. A return value of `false` causes `BlobClick()` to abort.

If the advisory causes `BlobClick()` to abort, `BClickResult()` returns zero. The cast returned from `BClickCast()` is undefined unless `BClickResult()` returns a non-zero value.

If the advisory causes `BlobClick()` to abort when `BlobClick()` has dragged a blob somewhere, the drag is considered bad and zoomback occurs as appropriate according to the value of the bad drag and zoomback flags. This value is set with `SetBCZoomFlags()`.

There are several observations to be made about the order of messages received by advisory functions. These are stated below without proof, but are important as they imply certain things about the state that advisories should consider themselves to be in at any given time.

(i) All attempted transactions begin with one of the messages `advDClick` or `advRClick`.

(ii) Transactions that are completable always receive a second message, which is either `advGlue`, `advUnglue`, `advXfer`, `advDup` or `advSwap`.

(iii) Not all transactions are completable. The user may fail to drag a blob onto an active receptor; the permissions may not all the transaction type; the advisory might have aborted `BlobClick()`.

(iv) (i) and (ii) imply that `advGlue`, `advUnglue`, `advXfer`, `advDUp` and `advSwap` messages are always immediately preceded by an `advDClick` or `advRClick` message.

(v) (iii) implies that `advDClick` and `advRClick` messages need not be followed by one of the `advGlue`, `advUnglue`, `advXfer`, `advDup` or `advSwap` messages.

(i) – (v) taken together lead to the result that the string of messages received by an advisory function may be described by the regular expression:

((advDClick|advRClick)+ (advGlue|advUnglue|advXfer|advDup|advSwap)*)*

where

`( )` indicates grouping

`x|y`* indicates alternation, i.e, *x* or *y

`+` indicates one or more of the preceding group

`*` indicates any number (possibly zero) of the preceding group

***** An example advisory would not be entirely inappropriate here. *****

3.7.3 “Donorless” Scenarios

3.7.4 Multiple Drag Regions

Multiple drag regions in receptor blobs may be simulated to some extent by clustering. Returning to the states and capital cities example, suppose we wish to display a small picture of the state and require that both the state name and the capital city name be associated with it. A receptor blob with two drag regions would be sufficient, but another way to do it would be to draw the state picture in the static region of one receptor blob, and cluster it with another receptor blob with no static region. The match set for each receptor blob would accept either the state name or the capital city name. If the donor blobs are set to single-use (maximum glue count of one), there is no possibility that a false match could occur, since if a name is dragged onto the drag region or either receptor, it cannot be dragged onto the other at the same time. Thus, if both drag regions are quiet, each must have one of the two desired names.

In general, a receptor blob with *n* drag regions may be simulated by *n*-clusters of single drag region receptors if the donor blobs are all single-use, by giving each receptor blob a match set consisting of every donor blob that is to be associated with the *n*-cluster.

This technique isn’t really very useful unless all drag regions are geometrically congruent.

3.7.5 Customizing the Blob Manager

3.7.5.1 Click Filter

3.7.5.2 Quiet Proc/Rand Proc