

```
#import "UndoManager.h"

// RK 10/31/94:Changed order of notification for willUndo(to be before execute:)
/*****
UndoManager
```

The UndoManager object is used to handle undo and redo on a document basis in an application. Objects in the application register changes to themselves by sending an "undo" message to an undoManager instance from within an undoable method. For Example:

```
- setRadius:(float)value
{
    // Give undoManager the target for this event, then send undo message.
    [[undoManager setUndoTarget:self] setRadius:radius];

    // Set the new value and return
    radius = value; return self;
}
```

The UndoManager keeps a group (or list) of records for each "undoable" change. Each record contains a pointer to an object, an action for that object to perform and the arguments for that action. Undo is achieved by executing the target/action information in these undo records.

To add several records to an undo group, enclose the registration messages between UndoManager's -beginUndoRecordGrouping and -endUndoRecordGrouping methods. Undo messages that are not surrounded by these methods are added to the undoList in a group by themselves.

Also since sometimes changes should be ignored (ie. all of the discrete movements of a shape in a drag loop should not be undoable, just the final overall move), the messages -disableUndoRegistration and -reenableUndoRegistration can be sent to control which events are accepted into an undo record. For example, you might call [undoManager disableUndoRegistration] on a mouseDown: to prevent methods that register themselves with the undoManager to register each mouse drag, then you would call [undoManager reenableUndoRegistration] on mouseUp: with another call that would undo the drag loop (ie, [undoManager moveSelectedObjectsTo:oldPoint]).

The UndoManager allows the User to set the number of levels of undo to maintain (-setLevelsOfUndo:). When the number of records exceeds this amount, the excess is removed from the end of the list.

The UndoManager can be instructed to free target or argument data when records are removed from the undoList (whether they fall off the end of the list or are executed during an undo command). This is achieved by the -freeUndoTarget and -freeUndoArgs methods. Additionally the -copyUndoArgs method will copy any pointers that are registered with the next message (it is then assumed that the UndoManager will free them).

To trigger an Undo record, simply call -undo. This places the UndoManager into a "redo" state. All messages that are registered during an Undo are kept in a "redo" list. Thus the undo manager provides for redo as well. Redo can be triggered with the -redo message. The redo list is emptied each time a new record is added to the undo list (because, redo should only work if there were previous undos).

Clients of the undoManager can be notified when an undo is executed by adding themselves to the UndoManager's delegate list(-addUndoDelegate:). The

only notification messages are `-undoManagerWillUndo:` and `-undoManagerDidUndo:` (called before and after an undo or redo). This notification is useful to do perform functions that all undo events may require, like calling `-display`.

Written by: Jeff Martin ([jmartin@bozell.com](mailto:jmartin@bozell.com))

You may freely copy, distribute and reuse the code in this example.

Don't even talk to me about warranties.

\*\*\*\*\*/

```
#ifdef hppa
```

```
#define TARGET(UndoRec)    (*(id *) (UndoRec->args+UndoRec->argSize-4))
```

```
#define SELECTOR(UndoRec) (*(SEL *) (UndoRec->args+UndoRec->argSize-8))
```

```
#else
```

```
#define TARGET(UndoRec)    (*(id *) UndoRec->args)
```

```
#define SELECTOR(UndoRec) (*(SEL *) (UndoRec->args+4))
```

```
#endif
```

```
#define BIT(x) (((unsigned int)1)<<(x))
```

```
#define BITSon(from, to) (((BIT(to) - 1)<<1) + 1) & !(BIT(from) - 1))
```

## @implementation UndoManager

```
- init
{
    [super init];
    undoList = [[Storage allocFromZone:[self zone]] initWithCount:0
                elementSize: sizeof(RecordGroup) description:@encode(RecordGroup)];
    redoList = [[Storage allocFromZone:[self zone]] initWithCount:0
                elementSize: sizeof(RecordGroup) description:@encode(RecordGroup)];
    delegateList = [[List allocFromZone:[self zone]] init];
    disabled = NO; undoing = NO; redoing = NO;
    levelsOfUndo = 42;
    target = NULL; freeArgsMask = copyArgsMask = 0;
    actionName = NXUniqueString("");
    return self;
}

/*****
- beginUndoRecordGrouping, - endUndoRecordGrouping
*****/
```

These methods allow the client of the UndoManager to place several undo records in one undo "event". Undo messages that are received while not surrounded by begin/end undoRecordGrouping are placed into the undo list individually.

\*\*\*\*\*/

- **beginUndoRecordGrouping**

{

RecordGroup recordGroup, \*ptr;

if((!recordGrouping++) && (!disabled)) {

    // Get the appropriate list

    id list = undoing? redoList :undoList;

    // Add new recordGroup to the list if there is not an empty one there

    ptr = (RecordGroup \*)[list **elementAt:0**];

    if(!ptr || [ptr->recordList count]) {

        recordGroup.**recordList** = **[[Storage allocFromZone:[self zone]]**

initCount:0

```

        elementSize: sizeof(UndoRecord) description:@encode(UndoRecord)]];
if(actionName)
    recordGroup.actionName = actionName;
else
    recordGroup.actionName = NXUniqueString("");
[list insertElement:(void *)&recordGroup at:0];

// If list has more elements than levelsOfUndo, remove one from end
if([list count] > levelsOfUndo){
    ptr = (RecordGroup *)[list elementAt:[list count] - 1];
    [self discardRecordGroup:ptr];
    [list removeElementAt:[list count]-1];
}
}
}
return self;
}

```

```

- endUndoRecordGrouping { recordGrouping--; return self; }

```

```
/******  
- disableUndoRegistration, - reenableViewRegistration
```

These methods allow the undomanager to ignore events. This is useful in a case like mouse dragging, where the application only wants the final move to be undoable and not all of the intermediate drags. These should be strategically placed in blocks and can be nested.

```
*****/  
- disableUndoRegistration { disabled++; return self; }  
- reenableViewRegistration { disabled--; return self; }
```

```
/******  
- setUndoTarget:object
```

This method sets the target of following undo records. This is necessary because the sender cannot be gleaned from the forward method. Returns self.

```
*****/  
- setUndoTarget:object { target = object; return disabled? NULL : self; }
```



```

// Setting the current name of RecordGroup
- setActionName:(const char *)aName
{
    RecordGroup *ptr;
    id list;

    actionName = NXUniqueString(aName);
    if(undoing || redoing){
        list = undoing? redoList :undoList;
        ptr = (RecordGroup *)[list elementAt:0];
        if(ptr){
            ptr->actionName = actionName;
        }
    }
    return self;
}

// Querying the Undo/Redo status for menu validation
- (const char *)lastUndoName
{

```

```
RecordGroup *ptr = (RecordGroup *) [undoList elementAt:0];
```

```
if(ptr && ptr->actionName)
    return ptr->actionName;
else
    return NULL;
```

```
}
```

```
- (const char *)lastRedoName
```

```
{
```

```
RecordGroup *ptr = (RecordGroup *) [redoList elementAt:0];
```

```
if(ptr && ptr->actionName)
    return ptr->actionName;
else
    return NULL;
```

```
}
```

```
- (NXAtom) currentActionName {return currentActionName;}
```

```

/*****
- freeUndoTarget, - freeUndoArgs, - freeUndoArgAt:(int)pos

```

These methods specify whether the UndoManager should free the target and/or its args when an undo record is executed or when it exceeds the number of levels of undo. These methods set bits in the freeArgsMask representing the argument position. Position 0 is the target. Position 1-15 represent the first 15 arguments to the method. Bits 0-15 in the freeArgsMask represent args to be freed when the record is discarded, while Bits 16-31 represent args to be freed when the record is executed.

```

*****/
- freeUndoTarget { freeArgsMask |= BIT(0)|BIT(16); return self; }
- freeUndoArgs { freeArgsMask |= ((-1) & (!BIT(0)) & (!BIT(16))); return self; }
- freeUndoArgAt:(int)pos { freeArgsMask |= (BIT(pos)|BIT(pos+16));return self; }

/*****
- freeUndoTargetOnRecordDiscard, - freeUndoArgsOnRecordDiscard,
- freeUndoArgOnRecordDiscardAt:(int)pos

```

These methods specify whether the UndoManager should free the target and/or

its args when an undo record exceeds the number of levels of undo. These methods set bits 0 - 15 in the freeArgsMask representing the argument position. Position 0 is the target. Position 1-15 represent the first 15 arguments to the method.

```

/*****
- freeUndoTargetOnRecordDiscard { freeArgsMask |= BIT(0); return self; }
- freeUndoArgsOnRecordDiscard { freeArgsMask |= BITSon(1,15); return self; }
- freeUndoArgOnRecordDiscardAt:(int)pos
{ freeArgsMask |= BIT(pos); return self; }

/*****
- freeUndoTargetOnRecordExecute, - freeUndoArgsOnRecordExecute,
- freeUndoArgOnRecordExecuteAt:(int)pos

```

These methods specify whether the UndoManager should free the target and/or its args when an undo record is executed. These methods set bits 16 - 31 in the freeArgsMask representing the argument position. Position 16 is the target. Position 17-31 represent the first 15 arguments to the method.

```

/*****
- freeUndoTargetOnRecordExecute { freeArgsMask |= BIT(16); return self; }
- freeUndoArgsOnRecordExecute { freeArgsMask |= BITSon(17,31); return self; }

```

```
- freeUndoArgOnRecordExecuteAt:(int)pos  
{ freeArgsMask |= BIT(pos+16); return self; }
```

```
/******  
- copyUndoArgs, - copyUndoArgsAt:(int)pos
```

copyUndoArgs and copyUndoArgsAt: tell the UndoManager to copy pointer arguments (like objects or strings) for convenience. It is assumed that the UndoManager will free them when the undoRecord is freed. Returns self.

copyUndoArgsFreeOnDiscard and copyUndoArgsFreeOnExecute are just like copyUndoArgs but they specify whether to free the args when the record is discarded or when it is executed. They all Return self.

```
*****/
```

```
- copyUndoArgs { copyArgsMask = -1; [self freeUndoArgs]; return self; }  
- copyUndoArgAt:(int)pos  
{ copyArgsMask |= BIT(pos); [self freeUndoArgAt:pos]; return self; }
```

```
- copyUndoArgsFreeOnDiscard  
{ [self copyUndoArgs]; [self freeUndoArgsOnRecordDiscard]; return self; }  
- copyUndoArgFreeOnDiscardAt:(int)pos
```

```

{ [self copyUndoArgAt:pos];[self freeUndoArgOnRecordDiscardAt:pos];return self;}

- copyUndoArgsFreeOnExecute
{ [self copyUndoArgs]; [self freeUndoArgsOnRecordDiscard]; return self; }
- copyUndoArgFreeOnExecuteAt:(int)pos
{ [self copyUndoArgAt:pos];[self freeUndoArgOnRecordDiscardAt:pos];return self;}

/*****
- copyUndoArgsForRecord: (UndoRecord *)undoRecord

    copyUndoArgsForRecord is used internally and does the actual work of copying
the pointers.
*****/
- copyUndoArgsForRecord: (UndoRecord *)undoRecord
{

    Method method = class_getInstanceMethod([TARGET(undoRecord) class],
        SELECTOR(undoRecord));
    int argCount = method_getNumberOfArguments(method);
    int i, offset;

```

```
char *type;

// Copy the target if requested
if(copyArgsMask & BIT(0)) TARGET(undoRecord) =
    [TARGET(undoRecord) copyFromZone:[self zone]];

// Copy each argument that the client requested
for(i=2; i<argCount; i++) if(copyArgsMask & BIT(i-1)) {

    // Get argument type and offset
    method_getArgumentInfo(method, i, &type, &offset);
#ifdef hppa
    offset += undoRecord->argSize;
#endif

    // If argument is a string copy with NXCopyStringBufferFromZone
    if(!strcmp(type, "*")) {
        char **stringPtr = (char **) (undoRecord->args+offset);
        *stringPtr = (*stringPtr)?
            NXCopyStringBufferFromZone(*stringPtr, [self zone]) : NULL;
```

```

    }

    // If argument is an object copy with -copyFromZone:
    if(!strcmp(type,"@",1)) {
        id *objectPtr = (id *) (undoRecord->args+offset);
        *objectPtr = [*objectPtr copyFromZone:[self zone]];
    }
}
return self;
}

/*****
- forward:(SEL)aSelector :(marg_list)argFrame

```

This method is how events are registered with the Undo Manager. When the UndoManager receives a message that it doesn't respond to, it assumes that this is an undo event and associates it with the current target. This undoRecord is then added to the undoList.

```

*****/
- forward: (SEL) aSelector :(marg_list) argFrame

```



```
{
    RecordGroup *ptr;
    UndoRecord undoRecord;

    // Register the event if we are not disabled and there is a current target
    if((!disabled) && target) {
        // Allocate a new undo record and set its fields
        int argSize = method_getSizeOfArguments(
            class_getInstanceMethod([target class], aSelector));
        undoRecord.freeArgsMask = freeArgsMask;
        undoRecord.argSize = argSize;

        // Make a copy of the arguments (copy deep if requested)
        undoRecord.args = NXZoneMalloc([self zone], argSize);
#ifdef hppa
        bcopy(argFrame - argSize, undoRecord.args, argSize);
#else
        bcopy(argFrame, undoRecord.args, argSize);
#endif
        TARGET((&undoRecord)) = target;
    }
}
```

```
if(copyArgsMask) [self copyUndoArgsForRecord:&undoRecord];

// Do implicit recordGrouping if needed
if(!recordGrouping)
    [[self beginUndoRecordGrouping] endUndoRecordGrouping];

if(undoing){
    ptr = (RecordGroup *)[redoList elementAt:0];
    [ptr->recordList addElement:(void *)&undoRecord];
} else {
    ptr = (RecordGroup *)[undoList elementAt:0];
    [ptr->recordList addElement:(void *)&undoRecord];
}

// Make sure that redo list is empty if in normal mode.
if(!undoing && !redoing)
    while([redoList count]){
        ptr = (RecordGroup *)[redoList elementAt:0];
        [self discardRecordGroup:ptr];
        [redoList removeElementAt:0];
    }
}
```

```

    }
}

// Reset registration masks
freeArgsMask = 0; copyArgsMask = 0;
return self;
}

/*****
- undo:sender

This message actually triggers the UndoManager to take the top undo record
and send out all of the events. The record is then freed.
*****/
- undo:sender
{
    RecordGroup recordGroup,*ptr;

    ptr = (RecordGroup *)[undoList elementAt:0];
    if(!ptr) return nil;

```

```
// Send WillUndo notification
[self sendNotification:@selector(undoManagerWillUndo:)];

// Get the last list that actually had undo records in it
while(ptr && ![ptr->recordList count]){
    [undoList removeElementAt:0];
    ptr = (RecordGroup *) [undoList elementAt:0];
}
recordGroup.actionName = ptr->actionName;
recordGroup.recordList = ptr->recordList;
[undoList removeElementAt:0];
// Beep if there was no record group
if(!ptr) NXBeep();

// Otherwise set redo flag, execute record group, reset redo flag & return
else { undoing = YES; currentActionName = recordGroup.actionName; [self
executeRecordGroup:&recordGroup]; undoing = NO; }

// Send DidUndo notification and return
```

```

[self sendNotification:@selector(undoManagerDidUndo:)];

return self;
}

/*****
- redo

This message actually triggers the UndoManager to take the top redo record
and send out all of the events. The record is then freed.
*****/
- redo:sender
{
    RecordGroup recordGroup,*ptr;

    ptr = (RecordGroup *)[redoList elementAt:0];
    if(!ptr) return nil;
    // Get the last list that actually had undo records in it
    while(ptr && ![ptr->recordList count]){
        [redoList removeElementAt:0];
    }
}

```

```

        ptr = (RecordGroup *) [redoList elementAt:0];
    }
    recordGroup.actionName = ptr->actionName;
    recordGroup.recordList = ptr->recordList;
    [redoList removeElementAt:0];
    // Beep if there was no record group
    if(!ptr) NXBeep();

    // Otherwise set redo flag, execute record group, reset redo flag & return
    else { redoing = YES; currentActionName = recordGroup.actionName; [self
executeRecordGroup:@recordGroup]; redoing = NO; }
    return self;
}

/*****
- (int)levelsOfUndo, - setLevelsOfUndo:(int)value

```

These methods allow programmatic manipulation for the levels of undo that an UndoManager maintains.

```

*****/

```

```
- (int)levelsOfUndo { return levelsOfUndo; }
- setLevelsOfUndo:(int)value { levelsOfUndo = value; return self; }
```

```

/*****
- addUndoDelegate:object, - removeUndoDelegate:object

```

These methods add and remove objects that are to receive undo notification. The two notifications that are currently supported are `undoManagerWillUndo:` and `undoManagerDidUndo` (sent before and after an Undo or Redo).

```

*****/
- addUndoDelegate:object { [delegateList addObject:object]; return self; }
- removeUndoDelegate:object { [delegateList removeObject:object]; return self; }
- sendNotification:(SEL)action
{
    int i; for(i=0; i<[delegateList count]; i++)
        if([[delegateList objectAtIndex:i] respondsToSelector:action])
            [[delegateList objectAtIndex:i] perform:action with:self];
    return self;
}

```

```

/*****
- discardRecordGroup:recordGroup;
- executeRecordGroup:recordGroup;
- freeUndoRecord:(UndoRecord *)undoRecord;

```

These methods free the individual records in a record group.

discardRecordGroup: calls freeUndoRecord:withMask: with the discard part of the freeArgsMask while executeRecordGroup: executes the records in the record group then calls freeUndoRecord:withMask: with the execute part of the the freeArgsMask.

freeUndoRecord walks through the record arguments free those that the freeMask requests to be freed. Returns self.

```

*****/
- discardRecordGroup: (RecordGroup *)group
{
    int i;

    // Free the records in the list, then the list
    for(i=0; i<[group->recordList count]; i++)

```



```

        [self freeUndoRecord:[group->recordList elementAt:i]
withFreeMask:freeArgsMask];
    group->recordList = [group->recordList free];
    return self;
}

- executeRecordGroup: (RecordGroup *)group
{
    int i;

    // Start undo record grouping, execute & free records & stop record grouping
    actionName = group->actionName;
    [self beginUndoRecordGrouping];
    for(i=[group->recordList count]-1; i>=0; i--) {
        UndoRecord *rec = [group->recordList elementAt:i];
#ifdef hppa
        [TARGET(rec) performv:SELECTOR(rec) :rec->args + rec->argSize];
#else
        [TARGET(rec) performv:SELECTOR(rec) :rec->args];
#endif

```

```

        [self freeUndoRecord:rec withFreeMask:(rec->freeArgsMask>>16)];
    }
    [self endUndoRecordGrouping];

    return self;
}

- freeUndoRecord:(UndoRecord *)undoRecord withFreeMask:(int)freeMask
{
    Method method = class_getInstanceMethod([TARGET(undoRecord) class],
        SELECTOR(undoRecord));
    int argCount = method_getNumberOfArguments(method);
    int i, offset;
    char *type;

    // Free the target if requested
    if(freeMask & BIT(0)) TARGET(undoRecord)= [TARGET(undoRecord) free];

    // Free anything that the args point to if requested
    for(i=2; i<argCount; i++) if(freeMask & BIT(i-1)) {

```

```

        method_getArgumentInfo(method, i, &type, &offset);
#ifdef hppa
        offset += undoRecord->argSize;
#endif

    // Free any strings or pointers with free() function
    if(!strcmp(type, "*", 1)) free(*(char **) (undoRecord->args+offset));

    // Free any objects with -free method
    else if(!strcmp(type, "@", 1)) [*(id *) (undoRecord->args+offset) free];
}

// Free the args
free(undoRecord->args);

return self;
}

/*****

```

- emptyUndoManager

emptyUndoManager removes all of the recordGroups from the undoList and redoList. An UndoManager client might do this when a document is saved. Returns self.

\*\*\*\*\*/

- **emptyUndoManager**

{

RecordGroup \*ptr;

while([undoList count]){

ptr = (RecordGroup \*)[undoList elementAt:0];

[self discardRecordGroup:ptr];

[undoList removeElementAt:0];

}

while([redoList count]){

ptr = (RecordGroup \*)[redoList elementAt:0];

[self discardRecordGroup:ptr];

[redoList removeElementAt:0];

}

```

    recordGrouping = 0;
    undoing = NO; redoing = NO;
    return self;
}

/*****
- free

    Frees each undo group in the undo and redo lists then it frees the undo and
redo lists.
*****/
- free
{
    [self emptyUndoManager];
    undoList = [undoList free];
    redoList = [redoList free];
    delegateList = [delegateList free];
    return [super free];
}

```

@end