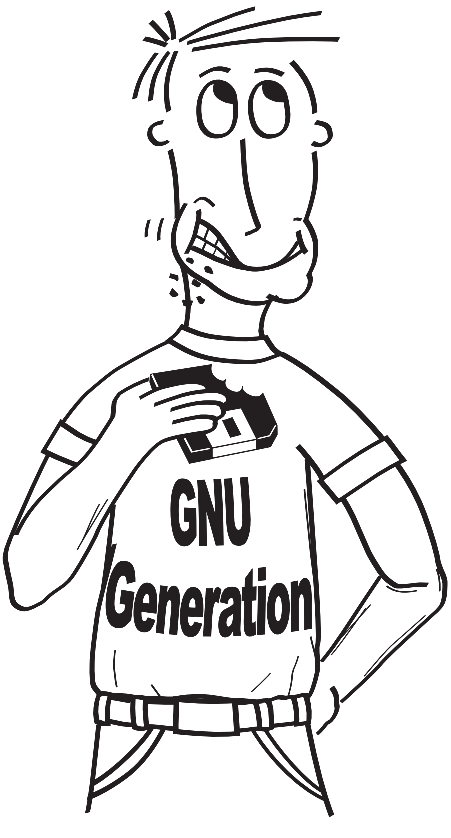
PROVIDING (REALLY) OPEN TECHNOLOGY.

# Inside LessTif

Information for the Hungry Programmer




Danny Backx
Mitch Miers
Chris Toshok
Harald Albrecht

ACKNOWLEDGEMENTS

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and we were aware of a trademark claim, the designations have been printed in caps or initial caps. We appologize for some designations being mangled during the process of typesetting. Really.

PIGS™ RATING

Absurd, Perverse, For Programmers Only. Attention: The information contained herein may cause damage to your opinions about the CSF and Graphical User Interfaces in general. You have been forewarned.

Edition BETA–IV, August 1997.

# Contents

# List of Tables

# List of Figures

# Foreword

When I first heard rumors of LESSTIF, I was already struggling for years with the famous M*TIF graphical user interface on different hardware platforms. Every new day unveiled another and – so to say – exciting feature not written down in the official documentation.

Needless to say that that official documentation was, and still is, lacking the *really* interesting aspects: writing not-so-trivial M*TIF widgets, how M*TIF's own geometry management utilities work, what the BaseClass stuff is for, and many other goodies. Ordinarily, this would be the time to "read the source, Luke". Unfortunately, the great creator of M*TIF – the CLOSED SOFTWARE FOUNDATION (or CSF for short) – reveals the source only to those willing to pay ("pay-per-view principle").

Fortunately, LESSTIF upheld the flag of "providing open technology" from its early beginning: the source is freely available under the terms of the GNU Public Library License. And it is common knowlegde that the source is also the source of information ("may the source be with you...").

Although standing on its own feet, LESSTIF provides deep insight not only into itself but also into the abyss of that other graphical user interface called "M*TIF". But browsing through the source and understanding all the intrinsic and intricate things inside LESSTIF isn't an easy task. You can easily get lost. Or to cite Mitch Miers and his law #37 of programming:

> *Real Programmers don't write documentation, they leave that to the tech writers.*
> *It was hard to code, it should be hard to understand.*

This is where "Inside LessTif" comes into play. Written by the creators of LESSTIF, this book provides a deep and thorough insight into this free M*TIF "clone". We show you all the mechanisms and smart improvements behind the scenes.

This book is definitely not a beginner's guide but is aimed at the programmer who has already a working knowledge of programming with – or should I say: fighting against – M*TIF. A basic knowledge about the Xt Intrinsics' widget class mechanism is necessary too.

I wish to thank especially all the authors for writing down their first-hand knowledge about LESSTIF. They had to cut off many valueable spare hours from their coding time to make this documentation real. Also my special thanks to Rob Blue, who has taken the (maybe daunting but nevertheless very important) part of proofreading.

Harald Albrecht

P.S.: If you find "Inside LessTif" to be useful (and we – the authors – are sure you will) then we ask you to donate a reasonable amount to a charitable institution.

# 1

## Synthetic Resources and Resolution Independence

Mitch Miers

## 1.1 Introduction

Synthetic resources are a mechanism included in Motif that allows a developer to modify resource values as collected by or assigned to the Xt resource mechanism. That is, if a user should want to find the value of an Xt resource, but M*TIF would rather that the user not see the true value, the synthetic resource mechanism allows the M*TIF developer to "fake out" the Intrinsics, and replace the true instance variable values with modified values. Alternatively, the toolkit may prefer to transform a user specified value into something more palatable by the toolkit.

The more common usage of synthetic resources is to support resolution independence (see figure 1.1). However, the toolkit developers also realized that the mechanism provided a way to protect "delicate" resources. For example, those that it would be dangerous for the user to change, or those that would upset the toolkit if they were unexpectedly modified.

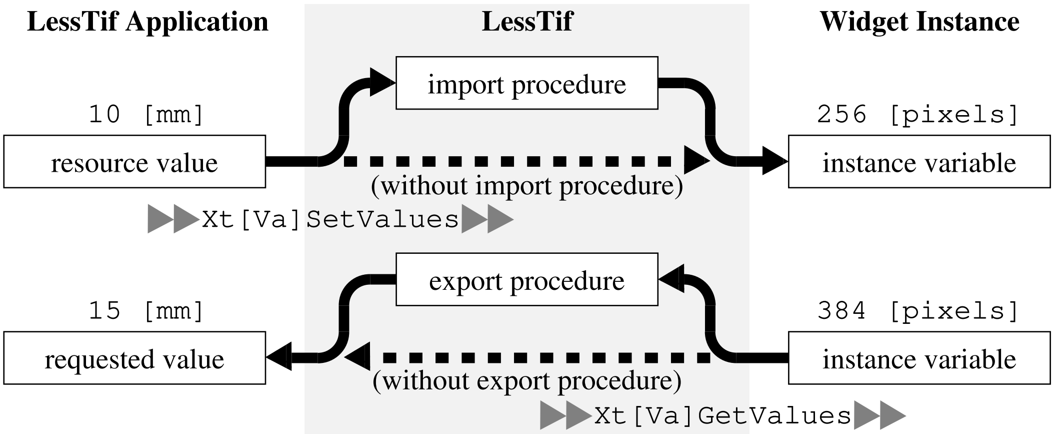



**Figure 1.1:** *Synthetic resources provide for data conversion as well as information hiding.*

The import and export direction is seen from M*TIF's point of view. A user specified resource value is *im*ported into some widget's instance variable. Au contraire, M*TIF *ex*ports a resource value whenever the user and/or application asks for a resource's value by calling `Xt[Va]GetValues()`.

## 1.2 The Implementation of Synthetic Resources

There are really five important classes in M*TIF: ExtObject, Gadget, Primitive, Manager and VendorShell. Nearly every Motif widget class inherits behavior from one of these. Some (`XmDialogShell` for example) inherit from more than one. While this may seem a statement of the obvious, it must be pointed out that the easiest way for a class to inherit behavior from a superclass is to subclass from the appropriate "Core" widget. Those aforementioned classes are the "Core" M*TIF classes (for objects, window-less widgets, core, constraint, and shell widgets respectively). Thus, if we provide the M*TIF "Core" widgets the ability to handle synthetic resources, all subclasses will inherit that behavior. In practice, the VendorShell class behavior is dictated by the X specification. Any special behavior related to the VendorShell must be specified in the VendorShell extension object, which is subclassed from ExtObj – thus the true core widget set contains only four classes.

There are only four class methods that deal with resource manipulation. They are the `initialize()` method, the `set_values()` method, the Constraint `set_values()` method, and the `get_values_hook()` method. Really there are two more, the `set_values_almost()` method, and the `set_values_hook()` method; the latter is obsolete, the former is in practice never used.

In terms of resource manipulation, the Core `set_values()` method and the Constraint `set_values()` method can be combined, by simply checking to see if the parent is a Constraint (read Manager) subclass. The only difference is where the resources are defined (self or parent). Second, the alterations that the `initialize()` method and the `set_values()` method perform based on the argument lists are identical; so we can use the same function for both. Therefore, we really have only two cases to handle – the case where the user specifies the value for a resource (`initialize()` and `set_values()`), and the case where the user wants to find the value of a resource (`get_values_hook()`).

Thus, there are two major sets of functions that implement the synthetic resource behavior: four import functions with the postfix `*ImportArgs()`, and five export functions ending with `*GetValuesHook()`. Both sets appear in `ResInd.c`. The prototypes are as follows:

```
void _XmExtImportArgs(Widget w, ArgList args, Cardinal *num_args);
void _XmPrimitiveImportArgs(Widget w, ArgList args, Cardinal *num_args);
void _XmGadgetImportArgs(Widget w, ArgList args, Cardinal *num_args);
void _XmGadgetImportSecondaryArgs(Widget w, ArgList args, Cardinal *num_args);
void _XmManagerImportArgs(Widget w, ArgList args, Cardinal *num_args);

void _XmPrimitiveGetValuesHook(Widget w, ArgList args, Cardinal *num_args);
void _XmGadgetGetValuesHook(Widget w, ArgList args, Cardinal *num_args);
void _XmManagerGetValuesHook(Widget w, ArgList args, Cardinal *num_args);
void _XmExtGetValuesHook(Widget w, ArgList args, Cardinal *num_args);
```

The import functions are called from the `initialize()` and `set_values()` methods in the relevant widget classes directly. The export functions are directly registered in the core `get_values_hook()` method.

Note that there is an extra function `_XmGadgetImportSecondaryArgs()` unique to Gadgets. It is a result of the "cache" subparts that nearly all Gadgets have. It provides the mechanism to import and export resources from the subpart.

The implementation of the functions is fairly ugly, and I wouldn't mind suggestions on how to improve their efficiency. Essentially, they use a doubly nested loop, going through the resource request list, and finding matches in the synthetic resource list. If a match is found, the import or export procedure in the synthetic resource structure is invoked.

I'll not go into further detail about how the M*TIF "core" widget classes hook into the synthetic resource mechanism. If you read the code to Gadget, Primitive, or Manager, it is fairly obvious (for example, just a call to _XmPrimitiveImportArgs() in Primitive's set_values_method).

One fairly important thing to note, is that the "core" class_part_initialize() methods combine a subclass's synthetic resources with all of the superclass's synthetic resources. This is done via _XmBuildResources(). This way, access to inherited synthetic resources get sped up because M*TIF doesn't need to search for them in the superclasses. If you subclass from a widget that has synthetic resources, but add no new ones, you'll end up with synthetic resources after the class_part_init() method has been called.

## 1.3 How to Use Synthetic Resources

The XmSyntheticResource structure looks suspiciously like the Xt resource structure, and in fact operates in much the same way. Here's the structure, and some associated information:

```
typedef enum {
    XmSYNTHETIC_NONE,
    XmSYNTHETIC_LOAD
} XmImportOperator;

typedef void (*XmExportProc)(Widget w, int offset, XtArgVal *value);
typedef XmImportOperator (*XmImportProc)(Widget w, int offset,
                                         XtArgVal *value);

typedef struct _XmSyntheticResource {
    String       resource_name;
    Cardinal     resource_size;
    Cardinal     resource_offset;
    XmExportProc export_proc;
    XmImportProc import_proc;
} XmSyntheticResource;
```

The import and export procedures (import_proc and export_proc respectively) each take three parameters: the widget for which the value has to be im- or exported, the offset of the resource value within the widget's instance structure, and finally a pointer. In case of a XmImportProc it points to the place where the new resource value can be found. In the other case (XmExportProc) the third parameter value points to the place where the converted result should be stored.

Here's an example right from Primitive.c:

```
/* Resources of the primitive class */
static XtResource resources[] = {
    /* ... */
    {
        XmNhighlightThickness, sizeof(Dimension),
        XtOffset(XmPrimitiveWidget, primitive.highlight_thickness),
        _XmFromHorizontalPixels, _XmToHorizontalPixels
    },
    /* ... */
};
```

This implements the resource XmNhighlightThickness as a synthetic resource. The import procedure _XmToHorizontalPixels as well as the export procedure _XmFromHorizontal-Pixels convert to/from the internal pixel based highlight thickness.

Note that the XmImportProc has a return type. This type is a clue to the synthetic resource implementation for whether or not the imported value should be loaded into the widget structure directly, or just into the argument list. Most would suspect that this should always go into the widget structure; however, remember that many manager widgets have associated children defined as part of their functionality (e.g., SelectionBox), and don't have direct visibility into some of the resources that can be specified.

Let's go into a little more detail about that. When an import procedure wants a value to go directly into a resource variable, it returns XmSYNTHETIC_LOAD (see figure 1.2). This is typically returned by such functions as _XmToHorizontalPixels(), which implements resolution independence into widgets. This has the effect of modifying directly the width (or x) instance variable of a widget.

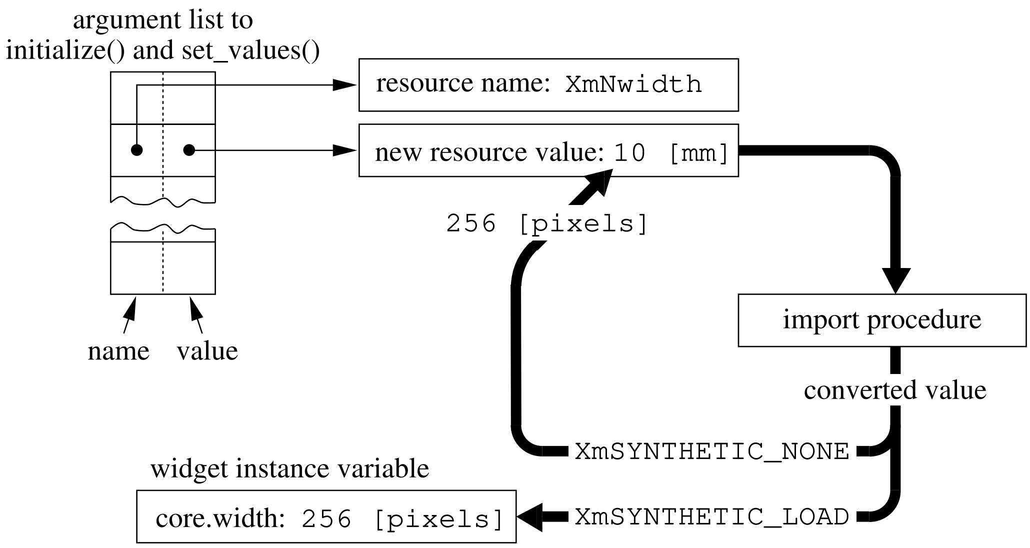



**Figure 1.2:** *The use of XmSYNTHETIC_LOAD and XmSYNTHETIC_NONE when importing synthetic resource values.*

However, consider the XmNlistItems in a SelectionBox. This import function typically wants to copy the XmStringTable provided by the user (as a "sensitive" resource that the user may or may not free). At any rate, the SelectionBox doesn't have visibility into this resource directly. The implementor has two choices: they can copy the XmStringTable and store it in the subpart in the import procedure (and return XmSYNTHETIC_LOAD), or they can simply make a copy of the XmStringTable, store it in the XtArgVal pointer, and return XmSYNTHETIC_NONE. In practice, they'll usually want to choose the latter, as they probably already have code in their set_values() or initialize() method to set the ListItems instance variable, and the synthetic resource mechanism will take care of the nuisance of making a copy first (at least, it will if that's how they've coded the import procedure). However, if they don't have code already, they can save some simply by coding the import procedure to do it for them; then they never need worry about the subpart (beyond checking to see if `set_values()` need return `True`).

For export procedures, much of the same logic applies, except that there is no return value, so only the resource values can be modified. The export procedures typically want to make a copy of "sensitive" resources, however (such as `XmNlabelString` in the Label widget – if the user frees this (as they should), it won't matter to the instance they got the value from. The same idea applies to things like `XmStringTable`'s in the List, etc. This can be a performance win, as well, as calls to `GetValues` can be avoided.

Also note that export procedures can play games. For example, consider the Gadget class, and what it should return for the value of `XmNtopShadowColor`. Take a look at the export procedure of the Gadget class for the `TopShadowColor` color:

```
static XmSyntheticResource syn_resources[] = {
    /* ... */
    {
        XmNtopShadowColor, sizeof(Pixel), XtOffset(XmGadget, object.parent),
        _XmGetParentTopShadowColor, NULL
    },
    /* ... */
};

static void
_XmGetParentTopShadowColor(Widget w, int offset, XtArgVal *value) {
    *value = XmParentTopShadowColor(w);
}
```

Here, we cheat. We know that the parent is a Manager, and that the `TopShadowColor` a Gadget uses is the `TopShadowColor` defined by the parent, so we can just return that. In this way, we can specify a value, even though we don't directly have an instance variable to define that value.

# 2

# Pandora's Box: The BaseClass Stuff

Harald Albrecht

## 2.1 Introduction

The "BaseClass stuff" – which is nicknamed after its implementation file `BaseClass.c` – consists mainly of three basic, interleaved parts that affect not only all widget classes of the LESSTIF toolkit but also "extend" the object-orientated design of the Xt Intrinsics. The three parts are the so-called "prehooks" and "posthooks", the "method wrappers", and finally the stacks for "widget extension data". Beside that, the BaseClass module contains some basic helper functions for messing around with "secondary resources" that are needed especially by gadgets. If you believe in redemption through Object Orientation, you should better skip this chapter completely and look out for other object(ive)s. You have been forewarned.

## 2.2 The Method Hooks

The BaseClass module provides hooks that are activated before and after a chain of widget methods have been called in superclass-to-subclass order. Prehooks and posthooks can be registered for arbitrary widget classes. They are available for:

- the `initialize()` method that initializes a single widget instance,
- the `class_part_initialize()` method that initializes a derived part within a class record,
- the `set_values()` method for setting resources, and finally
- the `get_values()` method that queries resources.

For example, the XmLabel class uses the hooks to do some pre- and post-processing when initializating a new widget instance – regardless of whether the new widget is a XmLabel or any subclass of (like XmPushButton). But also most gadgets make heavy use of the hook mechanism to manage their secondary objects that work as cache parts.

Unfortunately, while the concept of the hook mechanism looks straightforward at first glance, the implementation is definitely not. The difficulty is that the hook concept must be merged into the existing object orientated concept of the Xt Intrinsics without any chance of modifying and recompiling the Xt Intrinsics. Thus, the only way out is to twist method pointers.

But when twisting pointers, you can't simply hook up the first and last method pointer for any widget class, for example XmLabel. If some derived class, like XmPushButton, would hook up its method pointers too, the posthooks would interfere with each other. The reason is that the parameters supplied to the aforementioned four kinds of methods aren't enough to do the check whether the end of the method-chain has been reached and the posthook method must be called. Such a check could only be done using code generated at run-time – a practice which is definitely not portable across different systems. Later on, when looking at the wrappers for the `realize()`, `resize()` and `geometry_handler()` methods, you'll come across another solution to a quite similiar problem, but the solution there is even worse than what I'll present next.

To solve the hook problem, LESSTIF installs prehook *wrappers* as replacements for the methods `initialize()`, `class_part_initialize()`, `get_values()` and `set_values()` of the

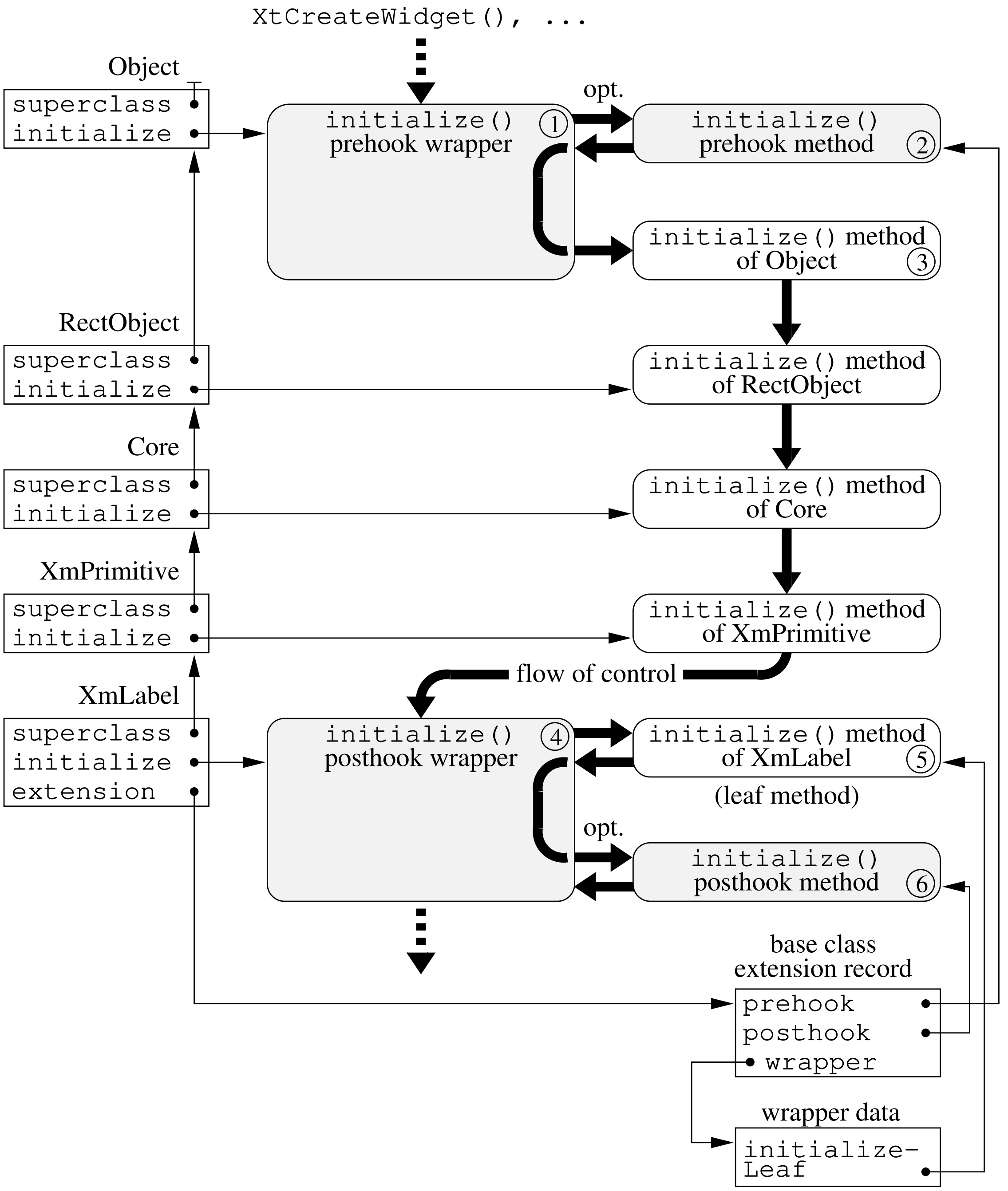



**Figure 2.1:** *The hook mechanism encloses superclass-to-subclass chained widget methods like* `initialize()`.

Object class. This insures that the hook wrappers are called first before any other subclass-method. This twisting of the method pointers is carried out by `_XmInitializeExtensions()` when the first VendorShell is created. The pointers to the old methods of the Object class are saved, so the prehook wrappers can chain them up. Note that there are no posthook (wrappers) installed at this time.

Now let us see what is happening lateron. I'll discuss this based on figure 2.1 for the `initialize()` method-chain of the XmLabel class. For this example, I assume that there is a prehook method ② as well as a posthook method ⑥ registered for the XmLabel class.

Whenever the `initialize()` method is invoked, the prehook wrapper ① is called first. In turn, it calls the prehook method ② – if there is one present for the widget class in question. Next, the prehook wrapper checks for the presence of a posthook method ⑥ – and if there is one, the prehook wrapper replaces the pointer to the final `initialize()` method ⑤ with a pointer to the posthook wrapper ④. The old pointer is saved using the "wrapper data stack" (you can find out more about this in the next section). Finally, the prehook wrapper calls the original `initialize()` method ③ of the Object class.

After this, the `initialize()` methods are called (as usual) in superclass-to-subclass order with the exception of the leaf method. If there is a posthook method registered, then – and only then – will the posthook wrapper ④ be called instead of the leaf method. The posthook wrapper restores the old method pointer, calls the leaf method ⑤, and finally activates the posthook method ⑥.

In one particular case the method-chains for `initialize()` and `set_values()` look different: whenever a widget is not a shell and it has a Constraint parent (or any subclass of) then the leaf method isn't the method of the widget's class but instead the parent's constraint leaf method (`constraint_initialize()` respective `constraint_set_values()`).

If you look close at the whole hook concept then it should be clear that this concept is error prone in certain situations when recursion is involved. The tradegy begins as soon as you enter the *same* prehook wrapper for a second time and the widget class of the second turn is a subclass of the widget class from the first visit to the prehook wrapper. In this case the *first* prehook wrapper method, which belongs to the superclass, is erroneously taken instead of the intended *second* prehook wrapper. Fortunately, there is no great chance to tap into this trap as long as you don't issue – for example – from the `get_values()` method of a XmLabel widget a call to `XtGetValues()` using another XmPushButton widget.

### 2.2.1 The Wrapper Data Stacks

The wrapper data stack plays an important role for the hook mechanism. For example, the prehook wrapper for `initialize()` etc. must keep the old pointer to the leaf method of the method-chain when it hooks in the posthook method instead. The old pointer is stored in a `XmWrapperDataRec` structure (figure 2.2). These wrapper data records are linked so that they form a stack for every single widget class. The head of the stack is accessible through a base class extension record. For the moment, I'll skip the base class extension record and come back to it in the next section.

The layout of a `XmWrapperDataRec` structure is as follows:

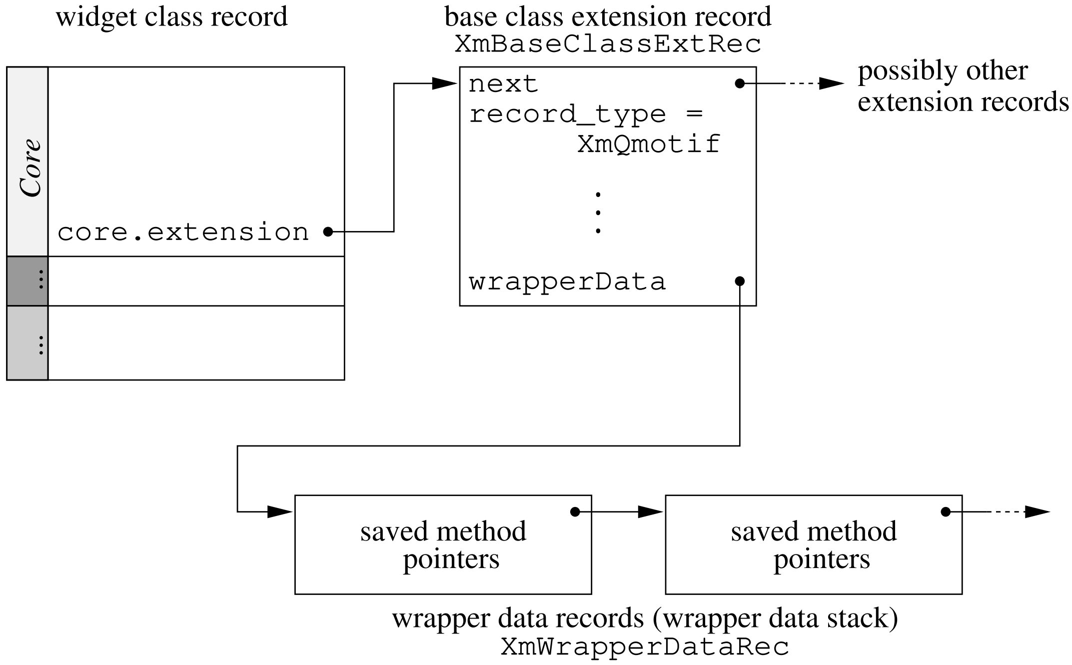



**Figure 2.2:** *The wrapper data stack keeps saved method pointers for a widget class.*

```
typedef struct _XmWrapperDataRec
{
    struct _XmWrapperDataRec *next;
    WidgetClass              widgetClass;
    XtInitProc               initializeLeaf;
    XtSetValuesFunc          setValuesLeaf;
    XtArgsProc               getValuesLeaf;
    XtRealizeProc            realize;
    XtWidgetClassProc        classPartInitLeaf;
    XtWidgetProc             resize;
    XtGeometryHandler        geometry_manager;
} XmWrapperDataRec, *XmWrapperData;
```

Now for a description of such a wrapper data record:

`struct _XmWrapperDataRec *next;`

Links to the next wrapper data record, so the data records form a stack.

`XtInitProc initializeLeaf;`
`XtWidgetClassProc classPartInitLeaf;`
`XtArgsProc getValuesLeaf;`
`XtArgsProc setValuesLeaf;`

These members keep the old pointers to the appropriate leaf method of the method-chain.

`XtRealizeProc realize;`
`XtWidgetProc resize;`
`XtGeometryHandler geometry_manager;`

These members keep the old pointers to methods with the same name. They belong to the "method wrappers" that are discussed in more detail in the next section.

`WidgetClass widgetClass;`
: Its use is not quite clear and it may be some leftover from an intended *full-featured* hook mechanism. At least it is currently superflous because you already know the widget class in order to get the most recent entry from the appropriate wrapper data stack.

Please note that the functions which work on wrapper data do **not** appear in any official header file (and thus not even in `BaseClassP.h`) and aren't accessible from the outside. This is true at least for M*TIF but unfortunately LESSTIF doesn't declare the wrapper functions `static`, so they are accessible. Despite this their use is strongly discouraged. I only talk about them here so you can understand their task within the BaseClass stuff. In M*TIF 2.0 the wrapper functions have lost their `_Xm` prefix to reflect their private state.

`XmWrapperData _XmPushWrapperData(WidgetClass wc);`
: Allocates a new wrapper data block on the heap ("free store" for all you "Ceplusplusists") and pushes it on the wrapper stack (aka LIFO) of the widget class specified by `wc`. If there is already a wrapper data block on the stack, the contents of the previous data block are copied into the new data block. `_XmPushWrapperData()` then returns a pointer to the most recent wrapper data block.

`XmWrapperData _XmPopWrapperData(WidgetClass wc);`
: Returns the most recent wrapper data block for the widget class `wc`. The data block is also removed from the wrapper stack, but the data is not being freed. It is the caller's responsibility to free the wrapper data block as soon as it is not needed any longer.

`XmWrapperData _XmGetWrapperData(WidgetClass wc);`
: Much the same as `_XmPopWrapperData()` but this time the wrapper data block is **not** removed from the wrapper stack of the widget class `wc`. If the wrapper stack should be empty at the time you call `_XmGetWrapperData()`, the function will create a blank wrapper data block on-the-fly and push it on the wrapper stack.

`void _XmFreeWrapperData(XmWrapperData data);`
: This is a LESSTIF-specific curiosity. It exists only for orthogonality reasons and is nothing more than a wrapper around `XtFree()`. Its only task is to free the memory occupied by the wrapper data structure pointed to by `data`.

### 2.2.2 The BaseClass Extension Record

Every widget class that wants to take part in the BaseClass game of hooks and wrappers must attach a data structure of type `XmBaseClassExtRec` to its class record. From now on I'll use the abbreviation "BCE record" (although its more an acronym) whenever I refer to a data structure of the type `XmBaseClassExtRec`. A BCE record can be used to add prehook and posthook methods to any widget class. Beside this, a BCE record also contains information about secondary objects, wrapper methods and other things right out of Pandora's box.

The layout of a BCE record is as follows:

```
typedef struct _XmBaseClassExtRec
{
    XtPointer             next_extension;
    XrmQuark              record_type;
    long                  version;
    Cardinal              record_size;
    XtInitProc            initializePrehook;
    XtSetValuesFunc       setValuesPrehook;
    XtInitProc            initializePosthook;
    XtSetValuesFunc       setValuesPosthook;
    WidgetClass           secondaryObjectClass;
    XtInitProc            secondaryObjectCreate;
    XmGetSecResDataFunc   getSecResData;
    unsigned char         flags[32];
    XtArgsProc            getValuesPrehook;
    XtArgsProc            getValuesPosthook;
    XtWidgetClassProc     classPartInitPrehook;
    XtWidgetClassProc     classPartInitPosthook;
    XtResourceList        ext_resources;
    XtResourceList        compiled_ext_resources;
    Cardinal              num_ext_resources;
    Boolean               use_sub_resources;
    XmWidgetNavigableProc widgetNavigable;
    XmFocusChangeProc     focusChange;
    XmWrapperData         wrapperData;
} XmBaseClassExtRec, *XmBaseClassExt;
```

If you want to attach your own BCE record to a (may be self-written) M*TIF widget class, take these steps:

① Within your class record, let `core_class.extension` point to your BCE record. This pointer can be set at compile-time.
② Set the `version` member of your BCE record to the value `XmBaseClassExtVersion` (at compile-time). This symbol is defined as soon as you include `BaseClassP.h`. Currently `XmBaseClassExtVersion` has the value "2".
③ Initialize the `size` member with `sizeof(XmBaseClassExtRec)`.
④ Initialize the `record_type` member with `NULLQUARK`. This is a constant, defined in the header file `X11/Xresource.h`, and is here used as a dummy for compile-time. At run-time, within the `class_initialize()` method of your widget class you must assign the value of the global variable `XmQmotif` to the `record_type` member.
⑤ At run-time, in your widget's `class_part_initialize()` method, set the fast subclass bit, which represents your widget class, using `_XmFastSubclassInit()`. More on this below.

If you don't need the BaseClass' whistles and bells for your self-written new widget class and you subclass from any of M*TIF's widget classes, then you can skip the steps mentioned above. If you don't attach a BCE record to your widget class record you'll automagically inherit a BCE record from your widget's superclass.

Now let's look at each member of a BCE record in more detail:

`XtPointer next_extension;`

Used when you need to chain more than one core class extension. Otherwise initialize with `NULL`.

`XrmQuark record_type;`

Indicates the kind of extension record. Within the `class_initialize()` method of your widget class, assign the value of the global variable `XmQmotif` to `record_type`.

`long version;`

Must be initialized with the constant `XmBaseClassExtVersion`.

`Cardinal record_size;`

Must be set to `sizeof(XmBaseClassExtRec).`

`XtInitProc initializePrehook;`
`XtInitProc initializePosthook;`

Pointers to the prehook and posthook methods. The prehook is called right before the first `initialize()` method triggers – that is, before Object's `initialize()`. The posthook is called after the last `initialize()` method has been called – usually this is the `initialize()` method of your (self-written) widget class.

Instead of specifying your own prehook or posthook method, you can use the identifiers `XmInheritInitializePrehook` and `XmInheritInitializePosthook`. In this case your widget class inherits the initialize hooks from its superclass. If you specify a `NULL` value as the pointer to a hook method instead, then you effectively disable for your widget class any initialize hook that may be present in your widget's superclass.

`XtSetValuesFunc setValuesPrehook;`
`XtSetValuesFunc setValuesPosthook;`
`XtGetValuesFunc getValuesPrehook;`
`XtGetValuesFunc getValuesPosthook;`

Much the same as the initialize hooks, but this time these hooks guard the `set_values()` and `get_values()` methods of your widget class. If you want to inherit one of these hooks simply specify the appropiate symbol `XmInherit[Set|Get]ValuesPrehook` or `XmInherit[Set|Get]ValuesPosthook`.

`XtWidgetClassProc classPartInitPrehook;`
`XtWidgetClassProc classPartInitPosthook;`

This is the fourth and last set of hooks, this time for hooking up the `class_part_initialize()` method. You surely can't imagine that there are two predefined symbols available to be used for inheritance: `XmInheritClassPartInitPrehook` and `XmInheritClassPartInitPosthook`. What a surprise...

`WidgetClass secondaryObjectClass;`

Specifies the widget class that is used to hold the secondary resources. "Secondary objects" are used for two main reasons. First, they extend such widget classes that already existed before M*TIF and now need new resources (especially VendorShell). Second, the secondary objects work as caches for such gadget resources that are likely to be the same in many gadget instances. When you specify `XmInheritClass` you'll inherit the object class specified by the superclass.

`XtInitProc secondaryObjectCreate;`

Points to the method that creates a secondary object of the class specified in `secondaryObjectClass`. When you specify `XmInheritSecObjectCreate` you'll inherit the object creation method from the superclass.

`XmGetSecResDataFunc getSecResData;`

Points to a function that returns descriptions about secondary resources of a widget class. The officially documented function `XmGetSecondaryResourceData()` (see the man pages from the CSF) uses this function pointer when an application wants to know what secondary resources a widget class has.

`unsigned char flags[32];`

Must be initialized with zeros, that is `{0}`. The `flags` array represents a bit field of the so-called "fast subclass bits". Within the `flags` bit field each bit represents one M*TIF widget class. If a bit is set, then this widget class is a subclass of the widget class that correspnds to the flag bit. The class identifiers look like `XmPRIMITIVE_BIT` or `XmTEXT_FIELD_BIT`, and can be found in the header file `Xm/XmP.h`. Note that these identifiers are bit numbers rather than bit masks. Use `_XmIsFastSubclass()` to check whether a given widget class has a particular fast subclass bit set.

`XtResourceList ext_resources;`

– under construction –

`XtResourceList compiled_ext_resources;`

– under construction –

`Cardinal num_ext_resources;`

– under construction –

`Boolean use_sub_resources;`

– under construction –

`XmWidgetNavigableProc widgetNavigable;`

Points to a method that checks whether a widget is navigable. Depending on the navigability of the widget the method must return either `XmNOT_NAVIGABLE`, `XmTAB_NAVIGABLE` or `XmCONTROL_NAVIGABLE`. This method pointer is used by `_XmGetNavigability()` from the keyboard focus traversal code. When you specify `XmInheritWidgetNavigable` for `widgetNavigable` you'll inherit the method pointer from the superclass.

`XmFocusChangeProc focusChange;`

Points to a method that will receive notifications from the keyboard focus traversal code whenever the focus changes. When you set `focusChange` to `XmInheritFocusChange` you'll inherit the method pointer from the superclass.

`XmWrapperData wrapperData;`

Must be initialized with `NULL` and will be used lateron by the BaseClass stuff. During runtime, the `wrapperData` member points to the most recent entry of the wrapper stack for this widget class.

## 2.3 The Method Wrappers

In contrast to the hook methods, the method wrappers are a special set of wrappers for the widget methods `realize()`, `resize()` and `geometry_handler()`. These three wrappers are only responsible for a few tasks within the LESSTIF toolkit. Therefore the BCE record has no method wrapper hooks that could be used by widget writers. The duties of the method wrappers are:

- The wrapper of the `realize()` method is available only for the VendorShell widget class and any subclass of it. The wrapper first calls the original `realize()` method and then it activates all callbacks registered in the `XmNrealizeCallback` resource of the VendorShell extension object.
- The `resize()` wrapper is available for the ExtObject class and any subclass of it. It first calls the original `resize()` method. Afterwards, it calls `_XmNavigResize()` so the keyboard focus traversal mechanism stays in sync with the mess on the user's display.
- The `geometry_handler()` wrapper makes sure that the drag&drop mechanism doesn't interfere with geometry management.

To see how the method wrappers are implemented, take a look at figure 2.3 that shows this exemplary for the `realize()` method wrapper.

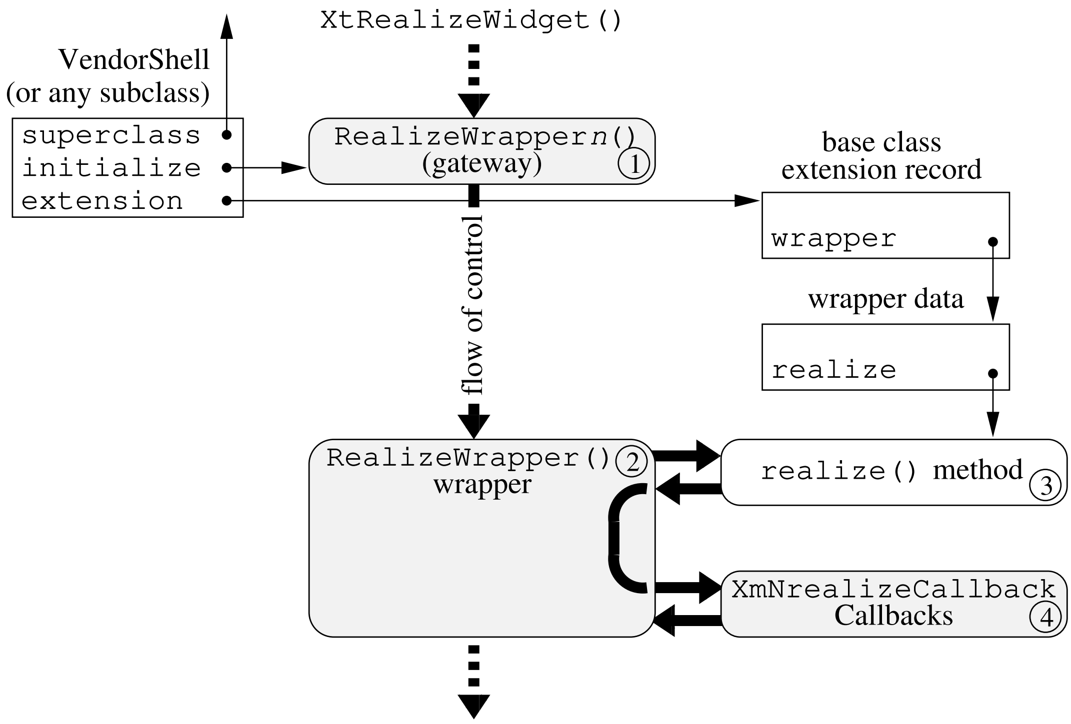



**Figure 2.3:** *The method wrappers enclose the non-chained widget methods like* `realize()`.

Unfortunately, just redirecting the `realize()` method pointers of all the class records to a wrapper method is not enough. How should the wrapper know which (original) method to call later on? It can't guess this information just from the widget identifier it got as one parameter to the method call: the intended method could be either from the widget's class or from one of the superclasses. A smart solution would be to generate dynamically a short piece of machine code (called a "thunk" in M$-babble) that is called before the wrapper. This piece of code then would supply the missing information to the wrapper method. Unfortunately, this isn't portable coding and is therefore useless to a project like M*TIF or LESSTIF.

As a way out, M*TIF as well as LESSTIF use a set of predefined wrapper "gateways" (marked with a ① in figure 2.3) that are statically compiled into the toolkits. In the case of the `realize()` method there are eight wrapper gateways called `RealizeWrapper0()` through `RealizeWrapper7()`. They all just call the real `realize()` wrapper ② and supply to it an additional parameter that is the distance between the widget class the gateway belongs to and the VendorShell class. For example, the `realize()` gateway of the VendorShell widget class has a distance of zero, whereas the XmDialogShell widget class has a distance of two.

Armed with the distance, the wrapper can then call the right `realize()` method ③ by substracting the distance, that the gateway indicated, from the distance of the class of the current widget, then walking up the according number of superclasses, and fetching the old method pointer from the wrapper data record of that superclass.

The description above applies to the `resize()` and `geometry_handler()` wrappers as well. The only difference is that the distance is now measured from the ExtObject class instead. In the case of the `geometry_handler()` the distance should probable have been measured from the Constraint class and not from the ExtObject class. But as M*TIF can't count, we can't either for compatibility reasons.

There are eleven gateway levels available for the `resize()` method and ten levels for the `geometry_handler()` method. So there is still room for larger widget class trees subclassed from XmPrimitive (six levels) and XmManager (only four levels due to the bug just mentioned) – but you have to live with these built-in limits, whether you want or not.

## 2.4 The Widget Extension Data

The widget extension data mechanism is a set of stacks that operate on a per-widget basis. You can push "extension data" (that is in the end a pointer to any block of memory) on a stack which belongs to a particular widget, peek at the data at any time, and finally pop it off the stack and free it.

LESSTIF uses the extension data mechanism for managing secondary objects and protocol objects. In order that LESSTIF doesn't mix up extension data intended for different purposes the extension data is "typed" or "tagged". Each widget has a different stack for each type/tag of extension data.

The heart of the widget extension data mechanism is a set of `XContexts` (you can find `XContexts` throughout the whole LESSTIF toolkit). With the help of the contexts you can associate a particular tagged stack of extension data with a widget (figure 2.4). The stacks for their part contain pointers to the extension data we're interested in.

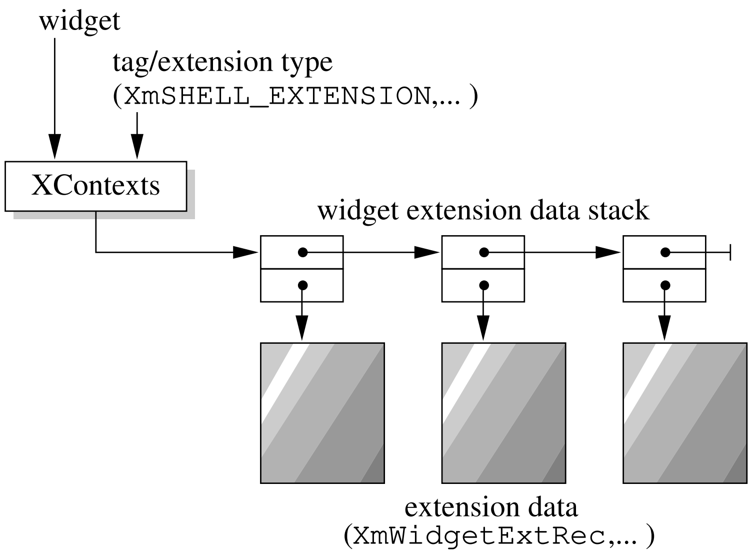



**Figure 2.4:** *A close look at the widget extension data mechanism.*

There are currently five extension data tags defined – see table 2.1. Gadgets use extension data that consists of a `XmWidgetExtDataRec` structure, which is tagged as a `XmCACHE_EXTENSION`. VendorShells use the same `XmWidgetExtDataRec` structure but tag them as a `XmSHELL_EXTENSION`.

| Identifier | Value |
|---|---|
| XmCACHE_EXTENSION | 1 |
| XmDESKTOP_EXTENSION | 2 |
| XmSHELL_EXTENSION | 3 |
| XmPROTOCOL_EXTENSION | 4 |
| XmDEFAULT_EXTENSION | 5 |

**Table 2.1:** *Tag identifiers for widget extension data*

A third use of the widget extension data mechanism is to manage a list of protocol objects in a `XmAllProtocolsMgrRec` structure tagged as `XmPROTOCOL_EXTENSION`. The protocol objectes are used to store information, so VendorShells can communicate with the window manager. The identifier `XmDEFAULT_EXTENSION` is only used by the (generic) ExtObject class and normally not needed. The use of the `XmDESKTOP_EXTENSION` is currently not known – it might be related to COSE/CDE.

As you can see from the previous explanations, the data structure `XmWidgetExtDataRec` plays an important role for the BaseClass stuff. So let us examine it in more detail:

```
typedef struct _XmWidgetExtDataRec
{
```

```
    Widget widget;
    Widget reqWidget;
    Widget oldWidget;
} XmWidgetExtDataRec, *XmWidgetExtData;
```

With Gadgets, the member `widget` of a `XmWidgetExtDataRec` points to the secondary object that acts as a cache for a set of the gadget's resources. The members `reqWidget` and `oldWidget` look suspiciously like the parameters from a `intialize()`, `set_values()` or `get_values()` method of a widget. In fact, they are used much the same, but this time they refer to a secondary object or a copy of the secondary object. Figure 2.5 illustrates this exemplary for the `set_values()` method.

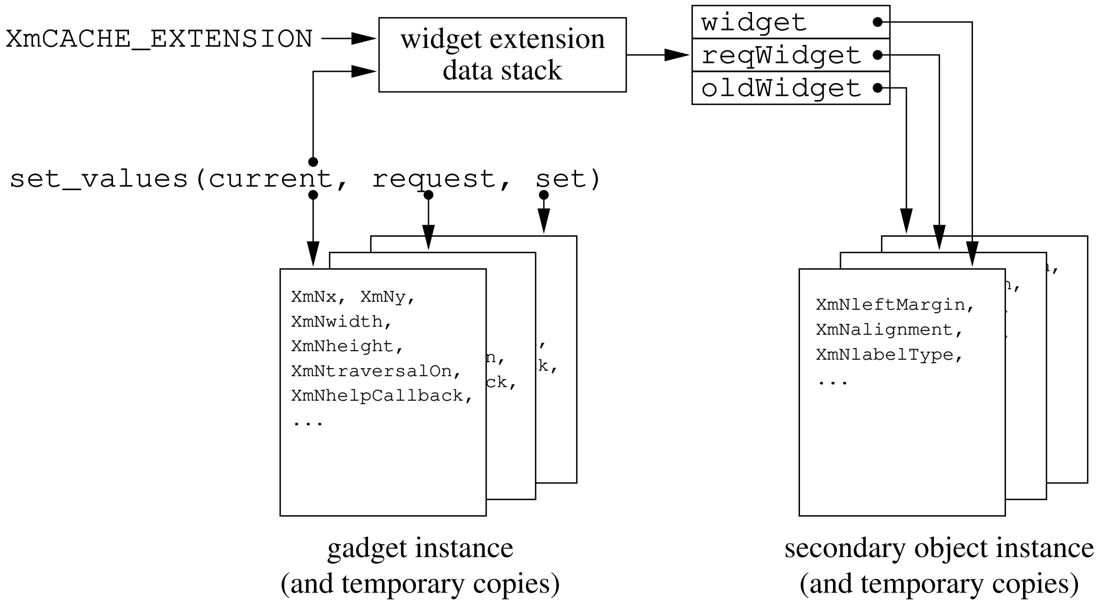



**Figure 2.5:** *The widget extension data mechanism helps to organize and manage secondary objects within some widgets methods.*

What you've just read about gadgets and their widget extension data applies to VendorShells almost the same. The only exception is that LESSTIF is lazy and in this case it creates the secondary object – which is a VendorShell extension object – using `XtCreateWidget()`. Thus it needs not to take care of `reqWidget` and `oldWidget`. Only the `widget` member in the `XmWidgetExtDataRec` is used to point to the VendorShell extension object (figure 2.6). A link (`ext.logicalParent`) within each instance of an ExtObject leads the way back to the VendorShell.

There are four – or rather three – functions within the BaseClass module that work with the widget wrapper extension data.

```
void _XmPushWidgetExtData(Widget widget, XmWidgetExtData data,
                          unsigned char extType);
```

Pushes an extension data record pointed to by `data` on the appropriate stack for extension

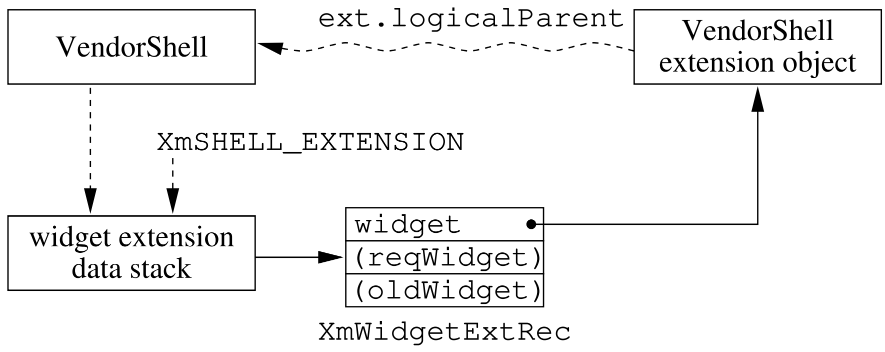



**Figure 2.6:** *The widget extension data represents the link between a VendorShell and its extension object.*

data of type `extType` of the widget specified by `widget`.

```
void _XmPopWidgetExtData(Widget widget, XmWidgetExtData *dataRtn,
                         unsigned char extType);
```

Pops off the most recent entry from the stack for extension data of type `extType` for the widget `widget`. The pointer to the data is returned in `dataRtn`. You are responsible to free the data when you don't need it any longer.

```
XmWidgetExtData _XmGetWidgetExtData(Widget widget,
                                    unsigned char extType);
```

Nondestructively fetches an extension data record. If there is currently no extension data of type `extType` on the stack of `widget` then the function returns `NULL`. Otherwise it returns a pointer to the most recent extension data record but does not pop it off the stack.

```
void _XmFreeWidgetExtData(Widget widget);
```

This one was surely implemented by some joker at the CSF. When called, it'll simply print out a warning that this function isn't supported. To be compatible, LESSTIF spits out an odd warning, too. Maybe `_XmFreeWidgetExtData()` once was intended to free all extension data that is associated with a particular widget. Maybe it was just a red herring.

## 2.5 Other Undocumented Stuff

There are a few undocumented functions and macros remaining that belong to the BaseClass stuff.

```
XmGenericClassExt *
  _XmGetClassExtensionPtr(XmGenericClassExt *listHeadPtr,
                          XrmQuark owner);
```

Can be used either to query any widget extension record you like or a BCE record. The parameter `listHeadPtr` must point to the **pointer** to the head of the list. Set `owner` to the kind of extension you're looking for. If you look out for a BCE record use the macro `_XmGetBaseClassExtPtr()` instead.

`_XmGetBaseClassExtPtr(wc, owner)`

This macro is defined in `Xm/BaseClassP.h` and tries to find a extension record of type `owner` and then returns a **pointer to the pointer** to the extension record. Typically, you'll use this macro to locate a BCE record of a particular widget class:

```
XmBaseClassExt *ext;
int             version;

ext = (XmBaseClassExt *)_XmGetBaseClassExtPtr(XtClass(w), XmQmotif);
version = (*ext)->version; /* access any structure member */
```

`Boolean _XmIsStandardMotifWidgetClass(WidgetClass wc);`

Checks whether the widget class `wc` is a standard widget class of M*TIF and then returns `True`. If the widget class to be tested isn't a standard widget class, the function returns `False`. The test is mainly based on the fast subclass bits within the base class extension records each widget class of M*TIF has. The widget class given in `wc` is considered to be a standard class if either:

– its superclass does not have a base class extension record (like XmPrimitve and XmManager),

– or the fast subclass bits of the widget class `wc` differ in at least one bit from the fast subclass bits of `wc`'s superclass. This way a self-written widget which is, for example, derived from XmManager and uses no new fast subclass bit, won't be considered to be from M*TIF's core widget set.

Only the fast subclass bits 0–191 belong to standard M*TIF widget classes, whereas the bits 192–255 are application-specific. Therefore the checks will include only the fast subclass bits in the range from bit 0 up to bit 191.

`Boolean _XmIsSlowSubclass(WidgetClass wc, unsigned int bit);`

If the widget class specified in `wc` has the fast subclass bit with the bit number `bit` set, this function returns `True`. Otherwise it returns `False`.

`_XmIsFastSubclass(wc, bit)`

This is not a real C function but rather a macro defined in `BaseClassP.h`. It checks whether the fast subclass bit with the bit number `bit` is set in the base class extension record of the widget class `wc`. If so, `_XmIsFastSubclass()` returns `True`. If the fast subclass bit is either not set or the widget class has no base class extension record, the macro returns the value `False`.

`_XmFastSubclassInit(wc, bit_field)`

This is not a real C function but rather a macro defined in `BaseClassP.h`. It sets the fast subclass bit with the bit number of `bit_field`.

`void _XmInitializeExtensions(void)`

This function initializes the BaseClass stuff. It is ordinarily called when the first VendorShell widget is created (typically within `XtAppInitialize()`) and sets up the `XmQmotif` quark. Finally, it hooks the prehook wrappers into the ExtObject class. Normaly, you would not need to call `_XmInitializeExtensions()` directly.

# 3

# Diverting User Input with Grabs

Harald Albrecht

## 3.1 Introduction

A “grab” is a mechanism that changes the way in which user input events – from the mouse or the keyboard – are reported. In addition to the Xlib grabs, which change the handling of user input within the X server (and which I’m not going to discuss here), the Xt Intrinsics add a second grab layer: the “Xt grabs”. They are similar to the Xlib grabs in that they change the way user input events are reported. But the Xt grabs work entirely within an application and therefore do not affect other applications using the same X server: the central event dispatch function `XtDispatchEvent()` is mainly responsible for the Xt grab mechanism.

Without going too much into the dirty details, it is important to know that the Xt Intrinsics maintain a grab list. If this list is not empty, only the widgets on the list, as well as their child widgets, will receive input events. Widgets can be added to the end of the grab list either as exclusive or non-exclusive. Only the widgets on the list from the most recently added exclusive widget to the end will receive input events. This way, adding a widget to the grab list exclusively will keep any user input events away from the widgets (and their children) that were already on the list. In the end, the grab list treats the widgets as a cascade. When you remove a widget from the list, all widgets that were added after it are removed, too.

M*TIF uses a different approach to the grab mechanism. Whereas the Intrinsics’ concept of a grab cascade is sensible for a set of cascading menus, it isn’t for modal dialogs. If there are two modal dialogs visible, popping down the older one should not remove the younger one from the grab list. Unfortunately, to achieve the desired grab behaviour M*TIF uses a completely undocumented grab layer, so the usual omnipotent disclaimer applies: continue reading at your own risk. All knowledge about M*TIF’s grab layer results from investigations done on the “living subject” with the help of a nice test program that can pretty-print various data trees to the console. The test program lives in `$(LESSTIF_ROOT)/testXm/extobj/test2.c`.

## 3.2 The Grab Layer and the Grab List

To be compatible with M*TIF’s grab concept, LESSTIF maintains its own grab list somewhat paralleling that inside the Xt Intrinsics. Whenever LESSTIF removes a grab from a widget by calling `XtRemoveGrab()` it puts back on all the widgets that are on its own list after the widget just removed.

LESSTIF manages the grabs on a per-display basis, thus you can find each grab list in the instance variable `modals` within the instance record of a `XmScreen` widget. The number of entries allocated for the list is stored in the `maxModals` instance member, whereas the `numModals` member – as usual – indicates how many entries currently are in use.

Two pure internal functions work on the private grab lists:

- `LTAddGrab()` adds a new widget to the grab list as either exclusive or non-exclusive.
- `LTRemoveGrab()` removes a widget from the grab list and puts back on all the widgets that were after it on the list.

Because the grab lists are mainly used to achieve dialog modality, and dialogs are eventually special shells, you can find the implementation of LTAddGrab() and LTRemoveGrab() in $(LESSTIF_ROOT)/libXm/Vendor.c.

Two additional functions – although not mentioned in the official documentation – are visible outside the Vendor.c module and provide access to M*TIF's grab layer (the interface functions are needed by the drag & drop mechanism and the menu system):

```
void _XmAddGrab(Widget wid, Boolean exclusive, Boolean spring_loaded);
void _XmRemoveGrab(Widget wid);
```

They both can be used just the same way as their counterparts from the Xt Intrinsics. It is very important **never to use** XtAddGrab() or XtRemoveGrab() as this will put LESSTIF's grab list out of sync with the Xt grab list.

Each entry of LESSTIF's grab list stores detailed information about a single grab:

```
typedef struct _XmModalDataRec {
    Widget                  wid;
    XmVendorShellExtObject ve;
    XmVendorShellExtObject grabber;
    Boolean                 exclusive;
    Boolean                 springLoaded;
} XmModalDataRec, *XmModalData;
```

The members wid, exclusive, and springLoaded of the XmModalDataRec structure record the values of the function parameters from the call to LTAddGrab() or _XmAddGrab(). This way, whenever removing a grab, LESSTIF can restore the grabs that were on the list after that grab.

When working with primary application modal dialogs, LESSTIF needs to remember such dialogs in grabber so it can remove, later on, any additional grabs it had to add in order to achieve the primary application modality. The grabber member, as well as ve, is not an ordinary widget ID but rather links to a so-called extension object of a vendor shell. I'll discuss this below in more detail.

### 3.2.1 Full Application Modal Dialogs

With a full application modal dialog on the display, the user must respond to it before she or he can do anything else in the application. To achieve this effect, LESSTIF must block out any user input events which are not meant for the full application modal dialog. LESSTIF therefore issues an exclusive grab on the shell widget of the dialog – see figure 3.1. In this figure, as well as in the next few ones, I'll leave out those child widgets of shell widgets that aren't shells themselves – otherwise the figures would be overcrowded.

In addition, in the following figures you'll see some entries on the grab list that are framed by thick edges. These entries form the "active Xt modal cascade": user input events are dispatched only to those widgets (as well as to their children) that are part of the active Xt modal cascade. The active Xt modal cascade always starts with the most recent exclusive grab.

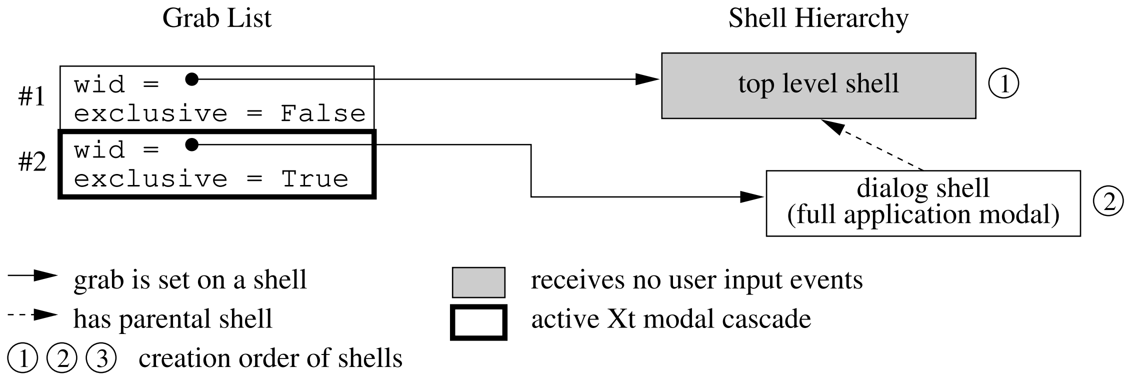



**Figure 3.1:** *Entries on the grab list after a full application modal dialog popped up.*

You may already be wondering why there is another (non-exclusive) grab in slot #1 of the list, just before our exclusive grab for the application modal dialog. This non-exclusive grab belongs to the top level shell of the application. LESSTIF installs such grabs on every VendorShell (or any subclass of) which does not serve as a popup shell (that is, as a dialog). Otherwise the top level shell of an application and any of its children won't receive input anymore as soon as a modeless dialog shows up, because modeless dialogs use grabs, too.

### 3.2.2 Modeless Dialogs

Although somewhat weird at a first glance, LESSTIF even installs a (non-exclusive) grab on the shells of modeless dialogs – see grab entry #3 in figure 3.2.

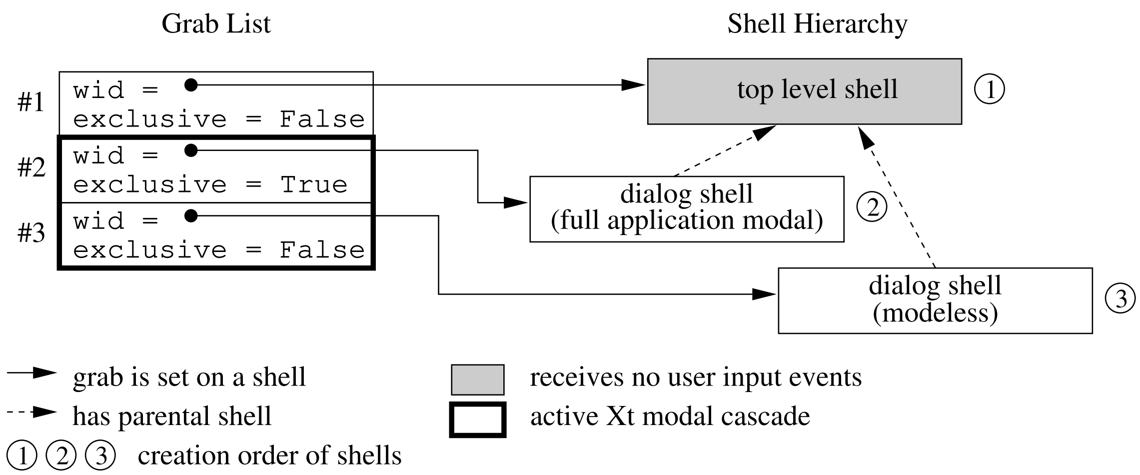



**Figure 3.2:** *Entries on the grab list after a modeless dialog popped up.*

The reason for this is that from LESSTIF's point of view the dialogs and their grabs don't form a cascade. Thus, if you pop up a modeless dialog whenever a full application modal dialog is already active, the new dialog should receive user input too – even if the modeless dialog is created as a child of the other dialog's parent and not as a child of the full application modal dialog.

### 3.2.3 System Modal Dialogs

System modal dialogs are much the same as full application modal dialogs. But with system modal dialogs, the user must respond to the dialog before doing anything else in *any* application. Because an exclusive Xt grab is only effective within the application, any LESSTIF application needs additional help from the window manager to make a dialog system modal. Currently, only `mwm` (and thus `lmwm`) provides the help needed. If there is no suitable window manager present, the system modal dialog behaves like any full application modal dialog.

In the case of a system modal dialog the grab list looks like the one presented in figure 3.2.

### 3.2.4 Primary Application Modal Dialogs

A primary application modal dialog (what a name!) is the fourth – and last – kind of LESSTIF's dialog types. Whenever such a dialog is visible, the user can interact with any dialog that is *not* a parent of the primary application modal dialog, just as with any other top level shell. Unfortunately, the behaviour just described isn't suited to the Xt grab mechanism at all. LESSTIF therefore has to do some work behind the scenes to get the desired grab behaviour: it must reissue all grabs that do not belong to any of the parental shells of our primary application modal dialog. A more elaborate example, figure 3.3, shows the grab list after a primary application modal dialog popped up.

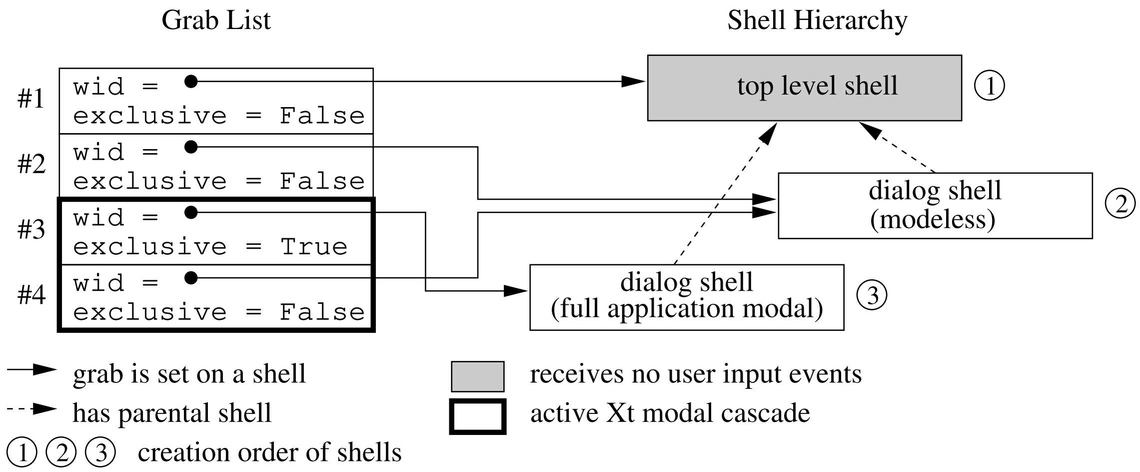



**Figure 3.3:** *Entries on the grab list after a full primary application modal dialog popped up.*

For some reason – yet not fully understood – M*TIFreissues the grabs for the shells, which aren't parents of the primary application modal dialog, only if `mwm` is present. Users of other window managers – like the famous `fvwm` – look (literally spoken) into the tube.

When popping down a primary application modal dialog, LESSTIF must get rid off all those grabs that once were necessary to get the desired grab behavior. In figure 3.3 the two grabs in the slots #2 and #4 are belonging to the *modeless* dialog ③. But only the grab from slot #4 must be removed together with the grab #3 that belongs to the primary application modal dialog.

This is where the `grabber` member mentioned above comes in. In the case of a primary application modal dialog the `grabber` links to an object (of class `XmVendorShellExtObject`) that belongs to the shell widget of the primary application modal dialog. All we then have to do inside `LTRemoveGrab()` is to remove all grabs from the list whose `grabber` belongs to the primary application modal dialog just popped down. In other cases of dialog grabs the member `grabber` links to the extension object of the same shell as referenced in `wid`.

## 3.3 Creating Dialog Shells the Right Way

There are two major ways of creating dialog shells:

- Either using `XtCreatePopupShell()` or one of the `XmCreate*Dialog()` functions. The latter ones just stand on the shoulders of `XtCreatePopupShell()` but hide this to some extend from the programmer. `XtCreatePopupShell()` also adds the shell widget just created to the parent's list of popup shells. All widgets (but neither gadgets nor objects) maintain such lists.
- Using `XtCreateWidget()` – and thus the same way as "ordinary" widgets are commonly created. If the parent of a shell is of class Composite (or any subclass of) then the (shell) widget is added to the parent's list of child widgets – which must be distinguished from the list of popup shells.

Although not recommended, `XtCreateWidget()` is widely used for creating dialog shells or shells in general (shame on me, too. Where's the sack and the ash?). Maybe the most prominent reason for this is that it makes widget creation more uniform. At the first look there seems to be no reason why to use `XtCreatePopupShell()` anyway. But M*TIF needs to know for its grab layer whether a shell widget serves as an ordinary top level shell (and thus has no modality) or as a dialog shell. In the latter case the `XmNdialogStyle` resource of a BulletinBoard-derived child of a dialog shell controls the dialog modality and eventually the grabs used.

The only way currently known to distinguish ordinary shells from shells working as popup dialogs is to check whether the shell is registered with the parent's list of popup shells. But this will fail miserably if the dialog shell was created using `XtCreateWidget()` instead of `XtCreatePopupShell()`. In that case the dialog shell first sets an unnecessary non-modal grab as soon as the shell is realized. Finally, when popping up the dialog, the `LTShellPopupCallback` callback sets the real (modal or non-modal) grab. Fortunately the first and unnecessary non-modal grab doesn't hurt very much – it just pollutes M*TIF's grab list. But the user won't notice that anyway.

## 3.4 Extending the VendorShell

The objects of the `XmVendorShellExtObject` class serve for two purposes. First, they provide storage for resources that are new with M*TIF's particular VendorShell. Second, they form a tree (or hierarchy) which shadows the instance hierarchy of all the shell widgets an application has. From now on I'll refer to this hierarchy as the "shadow shell tree". But first some more information about the rather less-spotted vendor shell extension objects.

By definition a VendorShell is a subclass of a WMShell and allows software vendors to provide new resources, class methods, etc. to support their custom window managers. So much for theory. In practice the spirit (idea) was willing but the flesh (the implementation) *is* still weak. If you would have to write your own VendorShell widget class that has additional instance variables ("resources") you would be in trouble – or rather those programmers writing their X applications using your new VendorShell. As several widget classes are derived from the VendorShell class (for example the TransientShell, the TopLevelShell and the ApplicationShell, see figure A.1 on page 117) these classes then would have to be recompiled. Otherwise the VendorShell part of an already compiled and older ApplicationShell would differ from that part of a new VendorShell. At this point the object-orientated approach simply can't handle the case where you need to replace a class *within* the class tree.

Historically, the CSF *needed* to add new resources to the VendorShell to support resolution independance, specific MWM functions and other "features". But extending the existing VendorShell was not desirable at all, as mentioned above, so the programmers at the CSF took another approach.

With M*TIF, every VendorShell widget (as well as every widget created from a subclass of VendorShell) is accompanied by a so-called VendorShell extension object. This VendorShell extension object (or "VSE object" for short) is a descendant of XmDesktopObject, XmExtObject, and thus finally of the Intrinsics' Object class (see figure 3.4).

The Object class was made public in X11R4 to enable programmers to use the Intrinsics' classing and resource handling mechanisms for things besides widgets. Within M*TIF, the VSE objects serve to hold all the instance variables (resources) that had to be added to the VendorShell class after the Xt Intrinsics had already been written down.

The starting point for all kind of extension object classes within LESSTIF is the XmExtObject class. This class provides for synthetic resources as well as external resources. In addition, every instance of a XmExtObject contains a pointer to its "logical parent" (resource `XmNlogicalParent`) that is the widget that owns the extension object (or that the extension object extends). You'll never find a XmExtObject in its logical parent's child list. One reason is that M*TIF seems to use some kind of self-crafted `XtCreateWidget()` when creating extension objects. A second reason is that LESSTIF currently uses a call to `XtCreateWidget()` when creating VSE objects. To cite Mitch, the affected logical parents keep "those pesky [extension] objects" out of the child list using special `insert_child()` and `delete_child()` methods.

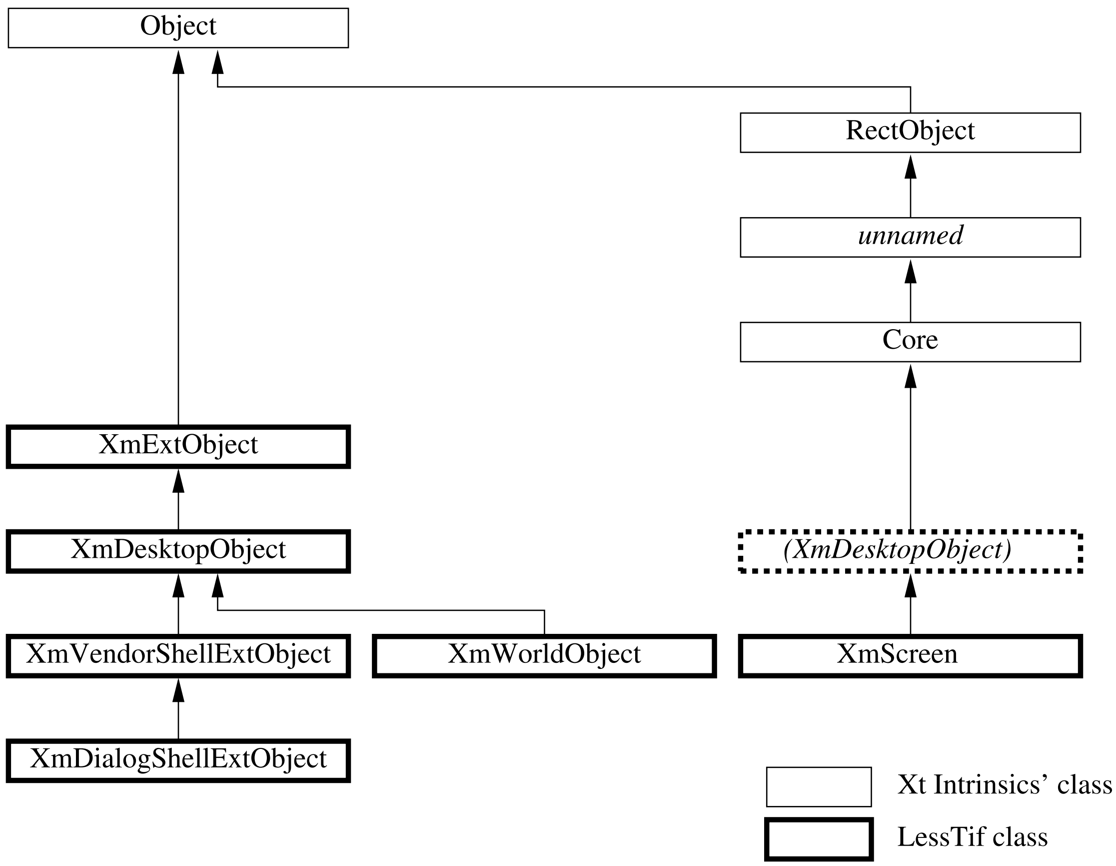



**Figure 3.4:** *Class hierarchy for extension objects.*

## 3.5 The Shadow Shell Tree

The VSE object (XmVendorShellExtObject) is subclassed from the XmDesktopObject class which in turn is subclassed from XmExtObject. Whereas the XmExtObject only knows of an associated "logical parent", the XmDesktopObject class introduces the concept of children to the extension objects (see figure 3.5).

The children of a XmDesktopObject (which must be either XmDesktopObjects again or a subclass of) are managed in a list much the same way Composite does. But as extension objects have no real parent (only a logical parent which is merely an associated widget) the XmDesktopObject class adds a new resource called `XmNdesktopParent`. It links to the parental object this particular XmDesktopObject is a child of. All the XmDesktopObjects form a "shadow tree" of the shell instance hierarchy. This tree is much like the well-known widget instance tree any application has – but this time any widgets that are not shells are left out.

At the top of the tree of XmDesktopObjects there is always a XmScreen widget. But as the XmScreen class is a *direct* subclass of Core and not of Composite it can't manage a list of children –

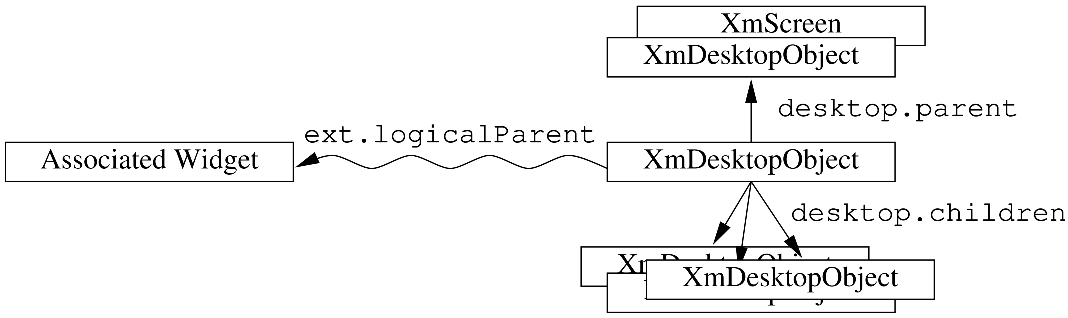



**Figure 3.5:** *The XmDesktopObjects form a tree of XmDesktopObjects with a XmScreen widget at the top.*

at least from the Xt Intrinsics' point of view. This is there the Desktop class part comes in a second time as "Nobody expects the Spanish Inquisition...". Oops – wrong movie.

In figure 3.4 you can see the XmDesktopObject (surrounded by dashed edges) reappearing at the branch between the Core class and the XmScreen class. At this point the principle of inheritance is somewhat broken as the XmScreen class is a *direct* subclass of Core. But XmScreen extends the instance record of the Core class not only by its own XmScreen-specific part but also by the XmDesktop-specific part.

This makes creating and inserting XmDesktopObjects ponderous because every such object must push its (desktop) parent to be inserted into the parent's list of children. But remember that the parent can be not only another XmDesktopObject but also a XmScreen widget. Thus, both XmDesktopObject as well as XmScreen provide methods through their desktop part within the class record for inserting (`insert_child()`) and deleting (`delete_child()`) a single child.

Whenever a XmDesktopObject is created, and its `XmNdesktopParent` resource is not `NULL`, it calls the `insert_child` method that is appropiate for the respective parent. As an example how this is done here is the `initialize()` method of the XmDesktopObject class (ripped of from `$(LESSTIF_ROOT)/libXm/Desktop.c`):

```
static void initialize(Widget request, Widget new_w,
                       ArgList args, Cardinal *num_args)
{
    Widget       desktopParent;
    XtWidgetProc insertChild;

    Desktop_Children(new_w)    = NULL;
    Desktop_NumChildren(new_w) = 0;
    Desktop_NumSlots(new_w)    = 0;
    desktopParent = Desktop_Parent(new_w);
    if ( desktopParent ) {
        if ( _XmIsFastSubclass(XtClass(desktopParent), XmSCREEN_BIT) ) {
            insertChild = ((XmScreenClassRec *) XtClass(desktopParent))->
                              desktop_class.insert_child;
        } else {
            insertChild = ((XmDesktopClassRec *) XtClass(desktopParent))->
                              desktop_class.insert_child;
```

```
        }
        if ( insertChild == NULL ) {
            _XmError(new_w,
                     "insert_child method of my desktop parent is NULL");
        }
        insertChild(new_w);
    }
} /* initialize */
```

The XmScreen widget at the top of the tree of XmDesktopObjects isn't yet the end of the road. Instead there's still another level represented by a XmDisplay widget (see figure 3.6). M*TIF creates such XmDisplay widgets for every display connection an application opens. The XmDisplay widget then gathers the various shadow shell trees via XmScreen widgets under its hood.

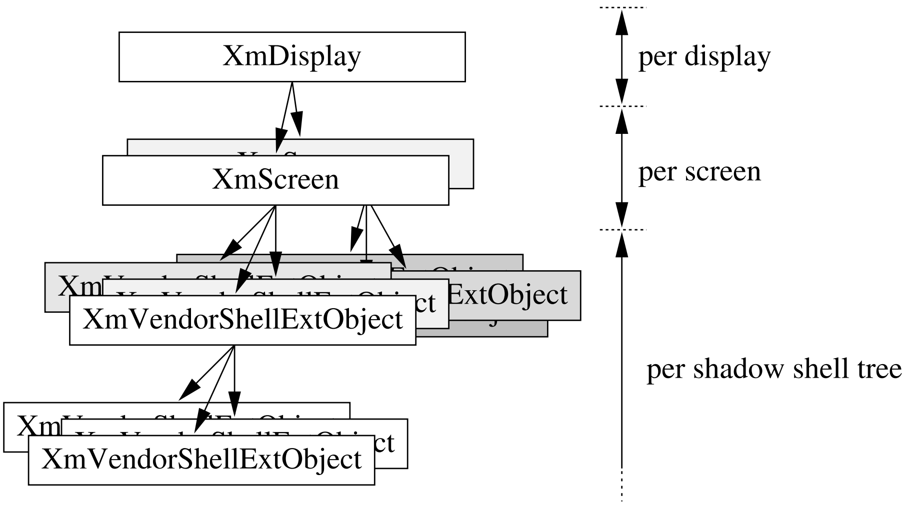



**Figure 3.6:** *The shadow shell tree is organized by XmScreen widgets which are in turn gathered under the hood of a XmDisplay widget.*

From time to time you need to look up the VSE object for a given VendorShell widget. For this you'll need the help of `_XmGetWidgetExtData()` from the BaseClass stuff. During initializing, every VendorShell widget creates a widget extension data record of type `XmWidgetExtDataRec` and pushes it on an extension data stack that LessTif maintains for some widget (and gadget) classes. The member `widget` of such a widget extension data structure finally points to the VSE object for the VendorShell. The following code excerpt shows how to find a VSE object using `_XmGetWidgetExtData()`.

```
Widget                  vendorShell;
XmVendorShellExtObject  ve;
XmWidgetExtData         extData;

/* vendorShell references a VendorShell widget */
extData = _XmGetWidgetExtData(vendorShell, XmSHELL_EXTENSION);
ve      = extData->widget;
/* ve now references the VSE object */
```

# 4

# Messy Geometry Management

Danny Backx
Mitch Miers

## 4.1 Introduction

This chapter describes the way in which LessTif widgets handle their geometry negotiations. As geometry management is a subject that has much to do with the Xt Intrinsics, part of this document describes (what we understand of) the Xt geometry model.

The Xt geometry model is based on geometry negotiations: every change that a widget wants to apply to its own geometry must first be approved by the widget's parent. The resources of the widget that are part of this geometry negotiation mechanism are:

- the position coordinates `x` and `y`,
- the `width` and `height`,
- and finally `border_width`.

A widget can request a geometry change by using `XtMakeGeometryRequest()` or `XtMakeResizeRequest()`. Whereas `XtMakeGeometryRequest()` can be used to change all five geometry resources mentioned above, `XtMakeResizeRequest()` only allows a widget to request a different size (width and/or height) and is in the end just a convenience function for such cases where the position of your widget doesn't matter.

In addition, LESSTIF has a convenience function called `_XmMakeGeometryRequest()` which calls `XtMakeGeometryRequest()`. But more on it below.

## 4.2 Making Geometry Requests

### 4.2.1 The Xt Intrinsics Way

The function prototype of `XtMakeGeometryRequest()` is as follows:

```
XtGeometryResult
    XtMakeGeometryRequest(Widget w,
                          XtWidgetGeometry *desired,
                          XtWidgetGeometry *allowed);
```

This function should be called from the widget which is passed as the first parameter. The second parameter describes the geometry that the widget would like to have. The last parameter returns the geometry that the widget got, in some circumstances. Finally, the result of the function is a value indicating whether the request has been granted.

The `XtWidgetGeometry` structure contains the five fields indicated above (position, extend, and border width), together with a bitset in which you can indicate which fields have been initialized. In the return parameter, the bitset will also indicate how much information is valid. A common mistake is to assume that the parent widget will always set the width and height values, and to just read those fields without looking at the flags.

The bitset field is called `request_mode`. It can be set using an OR of zero or more of the macros `CWX`, `CWY`, `CWWidth`, `CWHeight` and `CWBorderWidth`, each of which has exactly one bit set. A final bit `XtCWQueryOnly` is currently not used within LESSTIF. When set, the call to `XtMakeGeometryRequest()` will return a result without changing anything to the widget.

The result of `XtMakeGeometryRequest()` can have four values:

- `XtGeometryYes`: Means that the request has been granted.
- `XtGeometryDone`: The request has been granted, also it has been applied to the widget. According to "X Window System Toolkit" by Asente & Swick, a widget set should choose a policy: either use `XtGeometryYes` in all widgets, or use `XtGeometryDone`. In LESSTIF, we've chosen the `XtGeometryYes` approach.
- `XtGeometryNo`: You can guess this by now: the request has not been granted. Many manager widgets (subclasses of XmManager) in LESSTIF will return this when they get a request to change the `x` or `y` field (that is, the position) of a child widget.
- `XtGeometryAlmost`: This is a very useful but difficult return value. It means that the request has not been *completely* granted, and the `allowed` parameter returns a suggested geometry. If the widget takes the suggestion and calls `XtMakeGeometryRequest()` with that same set of values, the parent widget *must* allow this request. That's the hard part.

### 4.2.2 The LessTif Way

The function prototype of this geometry negotiation convenience function is:

```
XtGeometryResult
    _XmMakeGeometryRequest(Widget w, XtWidgetGeometry *desired);
```

This function is a wrapper around `XtMakeGeometryRequest()`. First, it asks the parent of the widget specified as the first parameter `w` for the new geometry (given in `desired`). If the parent's answer is `XtGeometryAlmost`, then `_XmMakeGeometryRequest()` takes a second round, asking the parent of `w` again, but this time using the geometry proposed by the parent. Then it returns.

Another programming mistake (introduced by Asente & Swick): given the Xt rule about `XtGeometryAlmost`, you could happily program a loop in which you keep calling `XtMakeGeometryRequest()` until the value is different from `XtGeometryAlmost`. The trouble is that this kind of a loop is only guaranteed to be finite if the parent widget(s) are bug-free. Need I say more?

`_XmMakeGeometryRequest()` detects this problem and is verbose about it: you'll see a warning, saying that a "Parent refused resize request" together with the name and class of the offending parent and the widget geometries in question. With LESSTIF, you might occasionally see this warning, because XmForm currently doesn't always grant a geometry that it just suggested...

If anybody wants a good exercise in understanding this document, she or he is invited to find this bug. Really. I'll only start tracking it myself when I have no serious bugs to attend to. A couple of beers (real or virtual? Ed.) can be had in Leuven, Belgium, by the first person to fix this.

## 4.3 Geometry Management and the Widget Methods

The basic cycle involved is:

① Calculate your *preferred size*.
② Ask for permission of your new preferred size using `XtMakeGeometryRequest()`.
③ Run your *layout* procedure.

For both primitives and composites, all the following rules for geometry management apply, except where otherwise stated. The rules mentioned in the subsections for the `geometry_manager()`, `change_managed()`, `insert_child()` and `delete_child()` methods apply only to composites. For Constraint widgets (and descendants), you should watch the rules mentioned for the `constraint_initialize()` and `constraint_set_values()` methods.

Just another note: if you have a composite widget, and this widget has either no children, or it doesn't has managed children (by the way, `_XmGeoCount_kids()` will return zero in the second case, but the GeoUtils are described in more detail in the next chapter), then you probably shouldn't bother in either method to compute the preferred size or the layout – mostly because you don't have anything to operate on. Instead you should probably return the current geometry in `query_geometry()`.

### 4.3.1 The initialize() Method

Don't do anything if you're a Composite (-derived) widget. You don't know enough yet (and you probably don't have any kids yet, unless the user created you with the `XtNchildren` resource: if the user does something like this, they're on their own). If you are a primitive widget (no pun intended...), you should know enough to do a basic layout.

### 4.3.2 The set_values() Method

If a value changed that should cause a layout change, go ahead and recompute the preferred size. Then, just set your width/height to the computed values: the Xt Intrinisics will automatically see a change made and call `XtMakeGeometryRequest()` on your behalf, so (ordinarily) don't do the layout step here.

If the request is granted, the Xt Intrinsics will automatically call your `resize()` method; that should be where the layout is done.

Rebuttal: There may be times where this isn't true. Some resources may require a Composite widget to re-layout. When doing so, there are a few warnings that should be noted.

- If the user set a resource that triggered a size adjustment, the resize request may not be honored by the parent. If it is not, and you re-layout, then be aware that you may have layed out to a geometry that isn't honored. This can cause (grossly understated) unexpected results. This case can (and does) happen in Label (and LabelGadget); that's why there is a call to the resize() method in expose() – to make sure the widget is displayed correctly. The same is true of CascadeButton (and CascadeButtonGadget).

- A superclass may have polluted the widget dimensions, or a resource that changed in a superclass may have altered the widget dimensions to cause a request that may not be honored. Laying out to this geometry may not be valid either.

Unfortunately, the intrinsics do not notify a widget if the resize request wasn't honored, so there's no way to do a proper job of it unless the expose() method calls the procedure that is responsible for the layout. Needless to say, this is not good for performance. One optimization that can be made is due to the nature of XtMakeGeometryRequest() (which will be discussed later): if the widget is not managed, or the parent isn't realized, then we can be sure that the resize request *will* be honored. In this case, we can blithely call our layout procedure and be sure that the request will be honored.

### 4.3.3 The resize() Method

You can't do any geometry negotiation here. You must take the size you currently have, and lay yourself out to this geometry. This method, in combination with set_values(), show the requirements of the two basic algorithms: one to compute the preferred size, and one to layout to a given size.

### 4.3.4 The realize() Method

For primitive subclasses, you probably want to realize your window, and then layout to it's geometry. For managers, it doesn't hurt to go through the "preferred size①" – "XtMakeGeometryRequest()②" – "layout③" cycle again, as things may have changed.

### 4.3.5 The query_geometry() Method

Call the routine that calculates the preferred size, and return the result if the request_mode is 0. Otherwise, return the usual rules of the request versus what you've computed.

### 4.3.6 The geometry_manager() Method

This is the trickiest one. The preferred-size-routine should take two parameters, instigator and instig_request, and use the values specified in instig_request when treating the instigator.

There are two additional rules that you should keep in mind: geometry_manager() doesn't get called if the parent isn't realized; it also doesn't get called if the child isn't managed (in both cases, the request is automatically granted). The method should then call `_XmMakeGeometryRequest()`. Doing this guarantees that either `XtGeometryYes` or `XtGeometryNo` is returned – you'll never see `XtGeometryAlmost` unless a composite has a bug.

Finally the layout function should be called. The layout function should take two additional parameters, the `instigator` and the `instig_request`. What to do next depends on the value that's going to be returned by the `geometry_manager()` method:

- If the result of the layout procedure on the instigator is `XtGeometryAlmost`, then no change should be made to the instigator (*or for that matter, to any other children in the composite*; this is an important point, and often alters the behavior of the layout method – doing a configure on a child when the geometry didn't change just wastes cycles). Instead, the reply geometry should reflect the layout computed for the instigator.
  **IMPORTANT**: if the instigator calls back with the geometry that you computed, the `geometry_manager()` method *must* return `XtGeometryYes`.

- If the result of the layout procedure on the instigator is `XtGeometryYes`, the *child's rectangle should be modified to reflect the geometry (this is important).* You must do this so that `XtMakeGeometryRequest()` will reconfigure the requesting child's window. This is different from `XtGeometryDone` in that `XtGeometryDone` implies that the window change was made by the parent, and we don't do that in LessTif. The GeoUtils and RowColumn do this now.

In the layout function, if the child being manipulated is *not* the instigator, then the child should be configured (normally using `_XmConfigureObject()`), if the return is `XtGeometryYes`.

### 4.3.7 The change_managed() Method

You've got to go through the complete cycle of "preferred size①" – "`XtMakeGeometryRequest()`②" – "layout③". The Xt Intrinsics don't help with any of this, so it's got to be explicit. Remember that you've no instigator here, so all managed children should be configured if a size change took place.

### 4.3.8 The insert_child() and delete_child() Methods

These methods are called when a child is added to a manager widget, or when a child is destroyed. Their use is particularly important in those manager widgets which keep information about their children in private data structures.

Note that these are unchained methods, which means they are not automatically called for all the superclasses of a manager widget. XmRowColumn's `insert_child()` needs to call XmManager's `insert_child()`, which in turn calls the one in its superclass.

### 4.3.9 The constraint_initialize() Method

Often, this method (if present) will not cause any geometry changes, but does offer an excellent time to capture information that will affect geometry management in the future. This includes things like the `XmNpositionIndex` resource (RowColumn), or `XmNpaneMinimum` and `XmNpaneMaximum` (PanedW).

### 4.3.10 The constraint_set_values() Method

Tricky. In this method, there can be so many interactions that the mind boggles. Quite often, resources that are set here may have *serious* implications for geometry management (like `XmNpositionIndex`), but it is difficult to know when a change should trigger a re-layout. In general, *all of the warnings for `set_values()` apply*.

### 4.3.11 The Geometry Management Helper Interfaces

There are two sets of two basic functions, that roughly have the following signatures. First, the ones for primitives:

```
PreferredSize( Widget w /* input */ );
Layout( Widget w /* input with side effects */ );
```

and for composites:

```
PreferredSize( Widget w,                           /* input */
               Widget instig,                      /* input */
               XtWidgetGeometry *instig_request, /* input */
               XtWidgetGeometry *preferred_geom  /* output */ );

Layout( Widget w,                           /* input */
        Widget instig,                      /* input */
        XtWidgetGeometry *instig_request, /* input/output */
        XtWidgetGeometry *preferred_geom  /* input */ );
```

For primitives, an example of the function to calculate the preferred size is `XmCalcLabelDimensions()`. The equivalent layout procedure would be the `resize()` method. Regardless, your layout function or preferred-size function can be modified for different behavior as appropriate to your class.

# 5

# Fun and Pain with the GeoUtils

Mitch Miers

## 5.1 Introduction

Recreating the behavior implemented in the GeoUtils was **a)** not fun, and **b)** really, really not fun. They are totally undocumented (as with almost everthing interesting the CSF ever did). What really kicked off the implementation of the GeoUtils was my discovery of John Cwikla's SmartMessageBox – without this gem, this work would have been impossible. A round of applause for this guy, please, and his intentions of "Furthering 'open software' into reality...".

The GeoUtils provide a mechanism by which BulletinBoard subclasses can automatically inherit geometry management for laying out their children. The good thing about the GeoUtils is that you get to specify `XtInherit*` for most of the class methods for a BulletinBoard subclass. The drawback is that you lose some flexibility (nevertheless a good tradeoff, because many of these functions are extremely difficult to write).

The mechanism provides for

- layout changes as a result of a child being managed or unmanaged (the BulletinBoard `change_managed()` method),
- layout at realize time (the BulletinBoard `realize()` method),
- layout for a resize (the BulletinBoard `resize()` method),
- a way to generically handle geometry queries from a parent (the BulletinBoard `query_geometry()` method),
- a way to generically handle geometry change requests from a child (the BulletinBoard `geometry_manager()` method),
- and finally a way to re-layout due to changes caused by a call to `Xt[Va]SetValues` (the subclass `set_values()` method).

The GeoUtils functions typically begin with `_XmGeo` (but not all do), whereas the corresponding BulletinBoard methods, for what the GeoUtils do normally, begin with `_XmGM`.

## 5.2 The BulletinBoard Class

There is one and only one clue in a BulletinBoard subclass that indicates that this class wants to use the GeoUtils – the `geo_matrix_create` method in the BulletinBoard class part (see `BulletinBP.h`). If this member is not `NULL`, the GeoUtils are activated for this subclass. The prototype for this method looks like:

```
typedef XmGeoMatrix (*XmGeoCreateProc)(Widget composite,
                                       Widget instigator,
                                       XtWidgetGeometry *desired);
```

The `composite` widget is (obviously) the BulletinBoard subclass. The `instigator` is used in the `geometry_manager` and `query_geometry` methods, as is the `desired` geometry (more on the use of those later).

### 5.2.1 The change_managed() and realize() Methods

Let's start with the two most similar cases: `change_managed()` and `realize()`. First take a look at BulletinBoard's `change_managed()` method (in `BulletinBoard.c` in the directory `$(LESSTIF_ROOT)/libXm`). Here, in the book, I will present only lines of concern as I talk about them (otherwise this book would get even larger).

Note that the first thing we do after we enter this method is look to see if the class record for the widget has a `geo_matrix_create()` member. If there is one, we call `handle_change_managed()` and return (more about this on page 43). If there isn't one, we proceed with generic BulletinBoard rules.

```
static void
change_managed(Widget w)
{
    Widget p;
    XmBulletinBoardClassRec *bb = (XmBulletinBoardClassRec *)XtClass(w);

    if (bb->bulletin_board_class.geo_matrix_create) {
        handle_change_managed(w, bb->bulletin_board_class.geo_matrix_create);
        return;
    }
```

Next, we call a `_XmGMEnforceMargin()`. This function ensures that the default BulletinBoard behavior of forcing children to be within the BulletinBoard margins is applied.

```
    _XmGMEnforceMargin(w, BB_MarginWidth(w), BB_MarginHeight(w), False);
```

Then we clear the old shadow, as what we may do could alter the way the shadow looks.

```
    _XmClearShadowType(w, BB_OldWidth(w), BB_OldHeight(w),
                       BB_OldShadowThickness(w), 0);
    BB_OldShadowThickness(w) = 0;
```

If we are realized, or our width or height is zero (usually indicating that this is the first child to be added), we call `_XmGMDoLayout()`. This function implements the generic BulletinBoard layout method.

```
    if (XtIsRealized(w) || XtWidth(w) == 0 || XtHeight(w) == 0) {
        _XmGMDoLayout(w, BB_MarginWidth(w), BB_MarginHeight(w),
                      BB_ResizePolicy(w), False);
    }
```

If we shrank, redraw the shadow (the `expose` method does this too, but...)

```
    if ((XtWidth(w) < BB_OldWidth(w) || XtHeight(w) < BB_OldHeight(w)) &&
```

```
        XtIsRealized(w)) {
        _XmDrawShadows(XtDisplay(w), XtWindow(w),
                       MGR_TopShadowGC(w), MGR_BottomShadowGC(w),
                       0, 0, XtWidth(w), XtHeight(w),
                       MGR_ShadowThickness(w), BB_ShadowType(w));
    }
```

Then, we record our width/height/shadow thickness.

```
    BB_OldWidth(w) = XtWidth(w);
    BB_OldHeight(w) = XtHeight(w);
    BB_OldShadowThickness(w) = MGR_ShadowThickness(w);
```

And finally, the required call to _XmNavigChangedManaged() that all Manager subclasses must do.

```
    _XmNavigChangeManaged(w);
}
```

If you read the code for `realize()` in `BulletinBoard.c`, you'll see almost identical code. The call to `_XmNavigChangeManaged()` isn't necessary in the `realize()` method, as is the test `XtIsRealized()` either. Instead the `realize()` method must chain up to its superclass' `realize()` method.

Now, let's take a look at the `handle_change_managed()` method that is called from BulletinBoard's `change_managed()` method. We start this function by checking if we are realized, or if our resize policy allows us to resize (i.e., not XmNONE). If either case is true, we set our desired width/height to zero; this is a cue to the GeoUtils to compute the desired size of this manager. If either case is false, we set the desired width/height to our current width/height; this cues the GeoUtils to lay out the manager to the current geometry (if possible).

```
static void
handle_change_managed(Widget w, XmGeoCreateProc mat_make)
{
    Dimension wd, ht, retw, reth;
    XmGeoMatrix geo;
    XtGeometryResult result;

    if (!XtIsRealized(w))
        wd = ht = 0;
    else if (BB_ResizePolicy(w) != XmNONE)
        wd = ht = 0;
    else {
        wd = XtWidth(w);
        ht = XtHeight(w);
    }
```

We then call the matrix create function. This function is crucial, as the data structures created tell the GeoUtils how to layout this particular widget.

```
geo = mat_make(w, NULL, NULL);
```

Next, we call `_XmGeoMatrixGet()`. This function essentially iterates through all the children we want to manage, querying each child (except the instigator) for the geometry the child desires. But note: this is *not* the same as all the managed children of this manager. If you forget to represent a child in the data structures when you create the matrix, that child won't be considered when you lay out the manager. Instead, at least in the case of SelectionBox and friends, the results are as specified in those class's documentation – usually "undefined" (in practice, they'll probably get piled up in the top lefthand corner of the manager).

```
_XmGeoMatrixGet(geo, XmGET_PREFERRED_SIZE);
```

Now the real work-horse routine in the GeoUtils is invoked – `_XmGeoArrangeBoxes()`. This function "parses" the data structure (the GeoMatrix) and lays out the children according to the rules defined by the matrix. Caveat: this function does *not* alter the children's geometry, but instead records the new geometry information in the `XmKidGeometry` structure contained by the GeoMatrix.

```
_XmGeoArrangeBoxes(geo, 0, 0, &wd, &ht);
```

At this point, `_XmGeoArrangeBoxes()` has computed the size of the manager as it would ideally like to be. The next code fragment checks the `BB_ResizePolicy` for a value of `XmRESIZE_GROW`. If the ideal size is less than the current size, we reject the change (because that would involve shrinking, and we should only grow). We then must re-layout the manager, by calling `_XmGeoArrangeBoxes()` with our current width and height.

```
if (BB_ResizePolicy(w) == XmRESIZE_GROW) {
    /* check the return against the original.  If the procedure would
     * like the BB to shrink, call again */
    if (wd < XtWidth(w) || ht < XtHeight(w)) {
        wd = XtWidth(w);
        ht = XtHeight(w);
        _XmGeoArrangeBoxes(geo, 0, 0, &wd, &ht);
    }
}
```

Now we look to see if any of the above calculations has indicated that the manager wants to resize (by comparing the computed width and height with the manager `XtWidth` and `XtHeight`). If no change is forthcoming, we just free the matrix and return. Otherwise we continue on.

```
if (wd == XtWidth(w) && ht == XtHeight(w)) {
    _XmGeoMatrixFree(geo);
    _XmNavigChangeManaged(w);
    return;
}
```

Okay, the manager wants to change size. We call `XtMakeResizeRequest()`, and ask our parent if we can change size. Eventually, the parent will respond with the size we can be (hopefully the size the manager wants, but we can compromise here).

```
    retw = wd;
    reth = ht;
    do {
        result = XtMakeResizeRequest((Widget)w, retw, reth, &retw, &reth);
    } while (result == XtGeometryAlmost);
```

The next fragment of code checks if a compromise was necessary, by evaluating whether the size our parent said we can be is the same as what we want to be. If the two don't match, we end up calling `_XmGeoArrangeBoxes()` yet again, to arrange our children to suit our parent.

```
    if (retw != wd || reth != ht)
        _XmGeoArrangeBoxes(geo, 0, 0, &retw, &reth);
```

Now that all the geometry calculation has been done, our parent is happy, and the manager is happy, we can go ahead and do `_XmConfigureObject()` calls on all our children. That particular job goes to the function `_XmGeoMatrixSet()`, which basically processes each `XmKidGeometry` box and configures the widget that the box represents.

```
    _XmGeoMatrixSet(geo);
```

If we've gotten this far, then we are pretty sure the manager's size has changed. If the manager has a shadow, now is the time to draw it (after erasing the old one).

```
    if (XtIsRealized(w)) {
        _XmClearShadowType(w, BB_OldWidth(w), BB_OldHeight(w),
                              BB_OldShadowThickness(w), 0);

        _XmDrawShadows(XtDisplay(w), XtWindow(w),
                       MGR_TopShadowGC(w), MGR_BottomShadowGC(w),
                       0, 0, XtWidth(w), XtHeight(w),
                       MGR_ShadowThickness(w), BB_ShadowType(w));
    }
```

We're done with the GeoUtils for now, so we can deallocate the matrix. Then we record our new width/height/shadow thickness. And finally, the required call to `_XmNavigChangedManaged` that all Manager subclasses must do.

```
    _XmGeoMatrixFree(geo);
    BB_OldWidth(w) = XtWidth(w);
    BB_OldHeight(w) = XtHeight(w);
    BB_OldShadowThickness(w) = MGR_ShadowThickness(w);
    _XmNavigChangeManaged(w);
}
```

The `realize()` case is identical to this one.

### 5.2.2 The resize() Method

The `resize()` method is *almost* identical to the two described above. The most significant difference is that we aren't supposed to talk back in this method, but accept whatever size we currently are, and lay ourselves out accordingly. Thus, that method is missing the calls that request the manager ideal size, and just does layout. Compare the `resize()` method to the `change_managed()` method above:

```
static void
resize(Widget w)
{
    XmBulletinBoardClassRec *bb = (XmBulletinBoardClassRec *)XtClass(w);
    Widget p;

    if (bb->bulletin_board_class.geo_matrix_create) {
        handle_resize(w, bb->bulletin_board_class.geo_matrix_create);
        return;
    }
    _XmGMEnforceMargin(w, BB_MarginWidth(w), BB_MarginHeight(w), False);
    _XmClearShadowType(w, BB_OldWidth(w), BB_OldHeight(w),
                       BB_OldShadowThickness(w), 0);
    BB_OldShadowThickness(w) = 0;
    if (XtIsRealized(w) || XtWidth(w) == 0 || XtHeight(w) == 0) {
        _XmGMDoLayout(w, BB_MarginWidth(w), BB_MarginHeight(w),
                      BB_ResizePolicy(w), True);
    }
    if ((XtWidth(w) < BB_OldWidth(w) || XtHeight(w) < BB_OldHeight(w)) &&
        XtIsRealized(w)) {
        _XmDrawShadows(XtDisplay(w), XtWindow(w),
                       MGR_TopShadowGC(w), MGR_BottomShadowGC(w),
                       0, 0, XtWidth(w), XtHeight(w),
                       MGR_ShadowThickness(w), BB_ShadowType(w));
    }
    BB_OldWidth(w) = XtWidth(w);
    BB_OldHeight(w) = XtHeight(w);
    BB_OldShadowThickness(w) = MGR_ShadowThickness(w);
}
```

You can also see the similarities in `handle_resize()` to `handle_change_managed()`:

```
static void
handle_resize(Widget w, XmGeoCreateProc mat_make)
{
    Dimension wd, ht;
    XmGeoMatrix geo;

    wd = XtWidth(w);
    ht = XtHeight(w);
    geo = mat_make(w, NULL, NULL);
    _XmGeoMatrixGet(geo, XmGET_PREFERRED_SIZE);
    _XmGeoArrangeBoxes(geo, 0, 0, &wd, &ht);
    _XmGeoMatrixSet(geo);
    if (XtIsRealized(w)) {
        _XmClearShadowType(w, BB_OldWidth(w), BB_OldHeight(w),
```

```
                            BB_OldShadowThickness(w), 0);

        _XmDrawShadows(XtDisplay(w), XtWindow(w),
                       MGR_TopShadowGC(w), MGR_BottomShadowGC(w),
                       0, 0, XtWidth(w), XtHeight(w),
                       MGR_ShadowThickness(w), BB_ShadowType(w));
    }
    _XmGeoMatrixFree(geo);
    BB_OldWidth(w) = XtWidth(w);
    BB_OldHeight(w) = XtHeight(w);
    BB_OldShadowThickness(w) = MGR_ShadowThickness(w);
}
```

### 5.2.3 The query_geometry() Method

For BulletinBoard, the `query_geometry()` method is probably the simplest – it does nothing on it's own behalf, but uses either the GeoMatrix (if a `geo_matrix_create()` method exists), or the generic method. This simplicity is deceiving; the complexity isn't visible in BulletinBoard – it's been shifted elsewhere.

```
static XtGeometryResult
query_geometry(Widget w, XtWidgetGeometry *proposed,
               XtWidgetGeometry *answer)
{
    XmBulletinBoardWidgetClass bbc = (XmBulletinBoardWidgetClass)XtClass(w);
    XtGeometryResult res;

    if (bbc->bulletin_board_class.geo_matrix_create) {
        return _XmHandleQueryGeometry(w, proposed, answer,
                                      BB_ResizePolicy(w),
                                      bbc->bulletin_board_class.
                                        geo_matrix_create);
    }
    res = _XmGMHandleQueryGeometry(w, proposed, answer,
                                   BB_MarginWidth(w), BB_MarginHeight(w),
                                   BB_ResizePolicy(w));
    return res;
}
```

### 5.2.4 The geometry_manager() Method

Similar to the `query_geometry()` method, the `geometry_manager()` method passes the buck on complexity. It looks somewhat like the `change_managed()`, `resize()`, and `realize()` methods, but does both less and more. It does less in terms of code in BulletinBoard, but the code in the GeoUtils is much more complex, and uses little of the code that the other three methods use. Also, the `geometry_manager()` method uses the "cache" variable in the BulletinBoard Widget instance, for repeated calls to `geometry_mananger()` when `geometry_manager()` returns `XtGeometryAlmost`.

```
static XtGeometryResult
```

```
geometry_manager(Widget w, XtWidgetGeometry *desired,
                 XtWidgetGeometry *allowed)
{
    Widget bb = XtParent(w);
    XmBulletinBoardWidgetClass bbc = (XmBulletinBoardWidgetClass)XtClass(bb);

    if (bbc->bulletin_board_class.geo_matrix_create) {
        return handle_geometry_manager(w, desired, allowed,
                                       bbc->bulletin_board_class.
                                         geo_matrix_create);
    }
    return _XmGMHandleGeometryManager(bb, w, desired, allowed,
                                      BB_MarginWidth(bb),
                                      BB_MarginHeight(bb),
                                      BB_ResizePolicy(bb),
                                      BB_AllowOverlap(bb));
}
```

If you read the `handle_geometry_manager()` code below, you'll see a similarity to the `change_managed()`, `realize()`, and `resize()` code above.

```
static XtGeometryResult
handle_geometry_manager(Widget w,
                        XtWidgetGeometry *desired, XtWidgetGeometry *allowed,
                        XmGeoCreateProc mat_make)
{
    Widget bb = XtParent(w);
    XmBulletinBoardWidgetClass bbc = (XmBulletinBoardWidgetClass)XtClass(bb);
    XtGeometryResult res;

    if (!(desired->request_mode & (CWWidth|CWHeight)))
        return XtGeometryYes;
    if (BB_OldShadowThickness(bb) != 0 ||
        BB_ResizePolicy(bb) != XmRESIZE_NONE) {
        _XmClearShadowType(bb, BB_OldWidth(bb), BB_OldHeight(bb),
                           BB_OldShadowThickness(bb), 0);
        BB_OldShadowThickness(bb) = 0;
    }
    res = _XmHandleGeometryManager(bb, w, desired, allowed,
                                   BB_ResizePolicy(bb), &BB_GeoCache(bb),
                                   bbc->bulletin_board_class.
                                     geo_matrix_create);
    if (!BB_InSetValues(bb) ||
        XtWidth(bb) > BB_OldWidth(bb) || XtHeight(bb) > BB_OldHeight(bb)) {
        if (XtIsRealized(bb)) {
            _XmDrawShadows(XtDisplay(bb), XtWindow(bb),
                           MGR_TopShadowGC(bb), MGR_BottomShadowGC(bb),
                           0, 0, XtWidth(bb), XtHeight(bb),
                           MGR_ShadowThickness(bb), BB_ShadowType(bb));
        }
    }
    BB_OldWidth(bb) = XtWidth(bb);
    BB_OldHeight(bb) = XtHeight(bb);
    return res;
}
```

Finally, we should discuss the `set_values()` method.

### 5.2.5 The set_values() Method

The `set_values` method is probably the most straight-forward BulletinBoard method there is. We aren't concerned about children wanting to change (the Intrinsics toolkit will use our `geometry_manager()` method if that is the case), we're only concerned with the user changing *us*. There is one issue, though – `set_values()` changes which change our geometry.

The reason for all this is that BulletinBoard, and subclasses of BulletinBoard, have a reasonably high number of children that are specified in instance variables. Rather than "ripple" the parent's geometry handler as children are changed, BulletinBoard and subclasses save them up until the `set_values()` method in the *class of the widget being changed* is called. The designers of M*TIF couldn't stop the `geometry_manager()` method from being called, but they added a mechanism that can control when the negotiation actually occurs.

The BulletinBoard widget has an instance variable called "`in_set_values`". This is a Boolean that is set when a `set_values()` method is invoked (keep in mind that the `set_values()` method is chained in super- to sub-class order), and cleared *almost* at the exit of the method. Right before the exit, the variable is cleared, and a test is used to see if a size update should be performed. If you look in BulletinBoard and subclasses in the `set_values` method, you'll see the following code fragment:

```
    if (need_refresh == True && XtClass(new) == xmBulletinBoardWidgetClass)
    {
        _XmBulletinBoardSizeUpdate(new);
        return False;
    }
```

For subclasses, replace the class check with a match for the subclass.

This trigger, in conjunction with an exception procedure (the `no_geo_request()` field in the GeoMatrix), keeps the `geometry_manager()` from handling possibly conflicting changes in the widget. During the `_XmHandleGeometryManager()` call, the `no_geo_request()` call is made to see if geometry negotiation should happen. Take a look at the `no_geo_request()` in SelectionBox:

```
Boolean _XmSelectionBoxNoGeoRequest(XmGeoMatrix _geoSpec)
{
    if (BB_InSetValues(_geoSpec->composite) &&
        XtClass(_geoSpec->composite) == xmSelectionBoxWidgetClass)
        return TRUE;
    return FALSE;
}
```

Should a geometry request come from a child during `set_values()`, the flag `BB_InSetValues()` will be `True`, and the negotiation will be delayed.

```
static Boolean
```

```
set_values(Widget old, Widget request, Widget new,
           ArgList args, Cardinal *num_args)
{
    BB_InSetValues(new) = True;
    /*
     * code block to handle set_values changes
     */
    BB_InSetValues(new) = False;

    if (XtWidth(new) != XtWidth(old) || XtHeight(new) != XtHeight(old)) {
        need_refresh = True;
    }
    if (need_refresh == True && XtClass(new) == xmBulletinBoardWidgetClass)
    {
        _XmBulletinBoardSizeUpdate(new);
        return False;
    }
    return need_refresh;
}
```

Note especially the call to `_XmBulletinBoardSizeUpdate()`. **This should be done in every BulletinBoard subclass that uses the GeoUtils**. This gives the manager class the opportunity to handle geometry changes in an instance's children that have occurred as a result of `set_values()`.

The code for `_XmBulletinBoardSizeUpdate` is as follows:

```
void
_XmBulletinBoardSizeUpdate(Widget w)
{
    XmBulletinBoardWidgetClass bbc = (XmBulletinBoardWidgetClass)XtClass(w);

    if (!XtIsRealized(w))
        return;
    if (bbc->bulletin_board_class.geo_matrix_create == NULL) {
        BB_OldWidth(w) = XtWidth(w);
        BB_OldHeight(w) = XtHeight(w);
        return;
    }
    if (!BB_OldShadowThickness(w) && BB_ResizePolicy(w) != XmRESIZE_NONE) {
        _XmClearShadowType(w, BB_OldWidth(w), BB_OldHeight(w),
                           BB_OldShadowThickness(w), 0);
        BB_OldShadowThickness(w) = 0;
    }
    _XmHandleSizeUpdate(w, BB_ResizePolicy(w),
                        bbc->bulletin_board_class.geo_matrix_create);
    if ((XtWidth(w) < BB_OldWidth(w) || XtHeight(w) < BB_OldHeight(w)) &&
        XtIsRealized(w)) {
        _XmDrawShadows(XtDisplay(w), XtWindow(w),
                       MGR_TopShadowGC(w), MGR_BottomShadowGC(w),
                       0, 0, XtWidth(w), XtHeight(w),
                       MGR_ShadowThickness(w), BB_ShadowType(w));
    }
    BB_OldWidth(w) = XtWidth(w);
    BB_OldHeight(w) = XtHeight(w);
    BB_OldShadowThickness(w) = MGR_ShadowThickness(w);
}
```

The function `_XmHandleSizeUpdate()` is very similar to the `change_managed()` method, in that it does layout computation, and requests a size change from the parent.

## 5.3 The Data Structures

Now that you have a passingly familiar with the basics, let's digress for a time and take a look at the data structures involved in the GeoUtils, as they should be understood before we talk about the implementation of an example subclass. There are three different data structures tangled up in the GeoUtils layout mechanism (figure 5.1): the `XmGeoMatrix` controls how the layout is to be performed, the `XmGeoMajorLayout` contains information about individual rows or columns of childen, and finally the `XmKidGeometry` records the geometry of a single child.

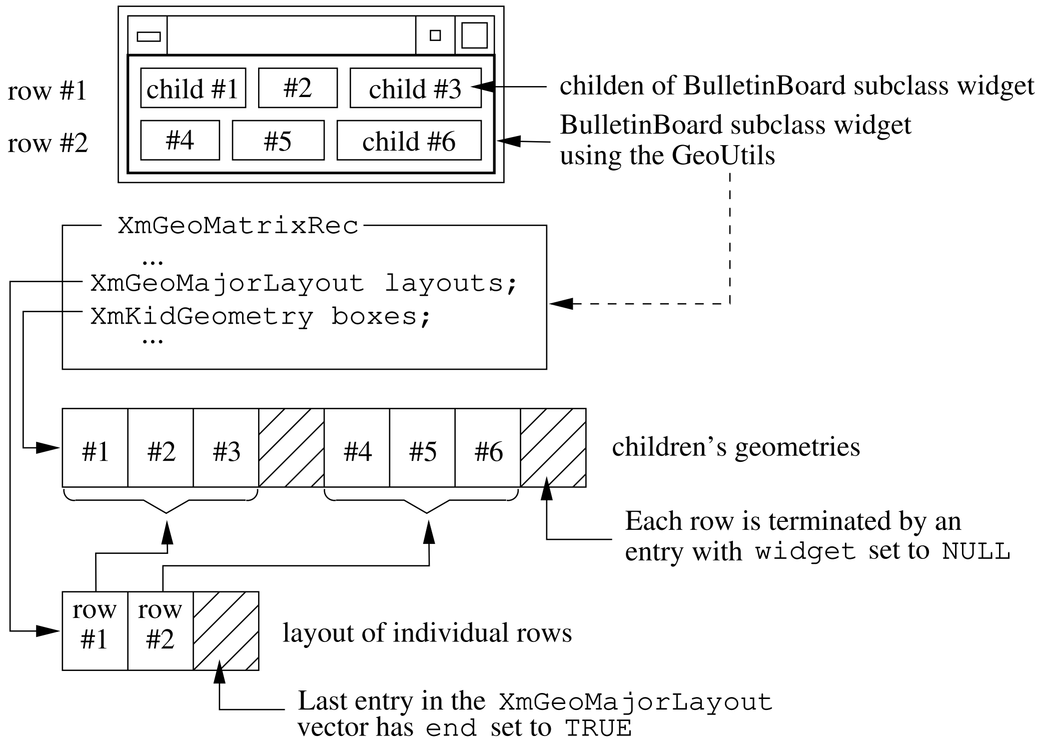



**Figure 5.1:** *Layout structures to mess around with when using the GeoUtils.*

### 5.3.1 The GeoMatrix

The layout of the GeoMatrix structure is as follows:

```
typedef struct _XmGeoMatrixRec {
    Widget composite;
```

```
    Widget instigator;
    XtWidgetGeometry instig_request;
    XtWidgetGeometry parent_request;
    XtWidgetGeometry *in_layout;
    XmKidGeometry boxes;  /* there is a NULL pointer at the end of each row */
    XmGeoMajorLayout layouts;
    Dimension margin_w;
    Dimension margin_h;
    Boolean stretch_boxes;
    Boolean uniform_border;
    Dimension border;
    Dimension max_major;
    Dimension boxed_minor;
    Dimension fill_minor;
    Dimension width;
    Dimension height;
    XmGeoExceptProc set_except;
    XmGeoExceptProc almost_except;
    XmGeoExceptProc no_geo_request;
    XtPointer extension;
    XmGeoExtDestructorProc ext_destructor;
    XmGeoArrangeProc arrange_boxes;
    unsigned char major_order;
} XmGeoMatrixRec;

typedef struct _XmGeoMatrixRec *XmGeoMatrix;

typedef void (*XmGeoArrangeProc)(XmGeoMatrix matrix,
                                 Position x, Position y,
                                 Dimension *width_inout,
                                 Dimension *height_inout);
typedef Boolean (*XmGeoExceptProc)(XmGeoMatrix matrix);
typedef void (*XmGeoExtDestructorProc)(XtPointer extension);
typedef void (*XmGeoSegmentFixUpProc)(XmGeoMatrix matrix, int command,
                                      XmGeoMajorLayout row_layout,
                                      XmKidGeometry kid_info);

enum {
    XmGEO_ROW_MAJOR,
    XmGEO_COLUMN_MAJOR
};
```

The GeoMatrix is the mother of all GeoUtils structures. In it, we have control information for how the layout is to be performed, info on each child, margin information, etc. Also, when you look at the GeoUtils, keep in mind that the developers intended for it to work both in row major and column major layout (i.e., up and down rows, and side to side columns). The comments in `XmP.h` indicate that they didn't get any further than row major layout, though. Column major layout hasn't even been implemented in M*TIF 2.0 – the comments are still there.

Let's look at each member:

`Widget composite;`

The BulletinBoard subclass instance that's currently using the GeoUtils.

`Widget instigator;`

If from `geometry_manager`, the child that requested a geometry change, or `NULL`.

`XtWidgetGeometry instig_request;`
: From `geometry_manager`, the change that the instigator requested, or `NULL`.

`XtWidgetGeometry parent_request;`
: If from `query_geometry`, the way our parent wants us to look, or `NULL`.

`XtWidgetGeometry *in_layout;`
: Used in the cases where multiple calls are made to `XtMakeResizeRequest()` or `XtMakeGeometryRequest()` from one of the children. There is a GeoMatrix "cache" instance variable in the BulletinBoard widget structure that gets used also. I don't think M*TIF's usage of this variable, and `instig_request` is *quite* the same as M*TIF's.

`XmKidGeometry boxes;`
: This member is used to keep layout information for each of the children of this manager. It is sort of a "cache" for the current and proposed geometry of each child. It is an array of structures: one structure for each child, each row of children separated by a structure whose child pointer is `NULL`.

`XmGeoMajorLayout layouts;`
: This is used to keep layout information for each row of children (especially things like whether each child in the row should be the same height, or the same width, or both, etc. More on this when we go throught the structure involved).

`Dimension margin_w;`
: The margin width of the manager.

`Dimension margin_h;`
: The margin height of the manager.

`Boolean stretch_boxes;`
: Whether or not children should be stretched to fill voids in the layout.

`Boolean uniform_border;`
: Whether or not the children should have the same `XtBorderWidth`. This can also be controlled on a row basis (the `XmGeoMajorLayout` has a `uniform_border` field, too. This value, if set, overrides the Layout structure's variable).

`Dimension border;`
: If `uniform_border` is true, the value that should be used for `XtBorderWidth`.

`Dimension max_major;`
: The maximum value of the major layout dimension. For row major layout, this would be the maximum computed width of all rows.

`Dimension boxed_minor;`
: For row major layout, this is the cumulative height of all the rows, not including fill.

`Dimension fill_minor;`
: For row major layout, this is the amount of fill space needed. In other words, the amount of "fill space" needed vertically between the rows.

`Dimension width;`
: This will hold the computed width of the manager.

`Dimension height;`
: This will hold the computed height of the manager.

`XmGeoExceptProc set_except;`
: A manager can override how the geometry of children are set by providing an override method here.

`XmGeoExceptProc almost_except;`
: I have no clue. Maybe a method that can be used if a parent says `XtGeometryAlmost` to a resize request?

`XmGeoExceptProc no_geo_request;`
: There are certain times when you want to avoid geometry negotiation for a while; usually in `set_values()`. This function is called from `_XmHandleGeometryManager` to determine if negotiation should really happen.

`XtPointer extension;`
: Extension data for use by the override methods. The GeoUtils don't do anything with this member directly.

`XmGeoExtDestructorProc ext_destructor;`
: A function that gets invoked when a GeoMatrix is freed, if the matrix has a non-`NULL` extension.

`XmGeoArrangeProc arrange_boxes;`
: An override method for arranging the children. If this is non-`NULL` most of the GeoUtils will not be used.

`unsigned char major_order;`
: An indicator for whether this matrix is row- or column-major. Currently only row-major is implemented. The values allowed here are `XmGEO_ROW_MAJOR` and (in principle) `XmGEO_COLUMN_MAJOR`.

### 5.3.2 The MajorLayoutRec

The next level of structures (actually, a union) control how the individual rows or columns are layed out.

```
typedef union _XmGeoMajorLayoutRec {
    XmGeoRowLayoutRec row;
    XmGeoColumnLayoutRec col;
} XmGeoMajorLayoutRec;

typedef union _XmGeoMajorLayoutRec *XmGeoMajorLayout;
```

The only member of interest is the XmGeoRowLayoutRec. Here's the layout for both; below, I'll describe what the fields mean for the RowLayoutRec – the fields of a ColumnLayoutRec (should it ever get implemented) are similar.

```
typedef struct {
    Boolean end;
    XmGeoSegmentFixUpProc fix_up;
    Dimension even_width;
    Dimension even_height;
    Dimension min_height;
    Boolean stretch_height;
    Boolean uniform_border;
    Dimension border;
    unsigned char fill_mode;
    unsigned char fit_mode;
    Boolean sticky_end;
    Dimension space_above;
    Dimension space_end;
    Dimension space_between;
    Dimension max_box_height;
    Dimension boxes_width;
    Dimension fill_width;
    Dimension box_count;
} XmGeoRowLayoutRec, *XmGeoRowLayout;

typedef struct {
    Boolean end;
    XmGeoSegmentFixUpProc fix_up;
    Dimension even_height;
    Dimension even_width;
    Dimension min_width;
    Boolean stretch_width;
    Boolean uniform_border;
    Dimension border;
    unsigned char fill_mode;
    unsigned char fit_mode;
    Boolean sticky_end;
    Dimension space_left;
    Dimension space_end;
    Dimension space_between;
    Dimension max_box_width;
    Dimension boxed_height;
    Dimension fill_height;
    Dimension box_count;
} XmGeoColumnLayoutRec, *XmGeoColumnLayout;

enum {
    XmGET_ACTUAL_SIZE = 1,
    XmGET_PREFERRED_SIZE,
    XmGEO_PRE_SET,
    XmGEO_POST_SET
};

/* fill modes for the GeoLayoutRec's below */
enum {
    XmGEO_EXPAND,
    XmGEO_CENTER,
    XmGEO_PACK
```

```
};

/* fit modes for the GeoLayoutRec's below */
enum {
    XmGEO_PROPORTIONAL,
    XmGEO_AVERAGING,
    XmGEO_WRAP
};
```

Now for a description of the `XmGeoRowLayoutRec`:

`Boolean end;`

If we have processed all the rows, this end flag will be true. In other words, if your widget has n rows of (child) widgets, your matrix will have $(n+1)$ row layout records, with the $(n+1)$ row having the end flag `True`. All other rows will have `end` set to `False`.

`XmGeoSegmentFixUpProc fix_up;`

Some rows might need special fixing after they've been laid out. For example, a separator in the SelectionBox should go the full width of the SelectionBox (as opposed to going from margin to margin). This `fix_up()` method allows such special cases to be handled. The only other special cases that I know about is a fixup for the MenuBar in SelectionBox and friends (extending the width so that it stretches for the full width of the parent, much like what is done for Separators).

`Dimension even_width;`
`Dimension even_height;`

These two members are overloaded. At the beginning of matrix processing, they are used as Booleans to indicate whether the children in this row should end up having the same width and height across the row. If they are `True`, they end up containing the maximum width and height of all the children in a given row, and then applied to each child after the max is computed.

`Dimension min_height;`

The minimum height for any given child in a row.

`Boolean stretch_height;`

Indicates if we can stretch (or shrink) the widgets in a row if the manager isn't quite the size we want.

`Boolean uniform_border;`

Assuming that the GeoMatrix didn't set its `uniform_border` member, this field indicates that this row should have a uniform border.

`Dimension border;`

Assuming that uniform_border (above) is true, the value of `XtBorderWidth` for the widgets in this row.

`unsigned char fill_mode;`

One of `XmGEO_EXPAND`, `XmGEO_CENTER`, or `XmGEO_PACK`. The only one of these I've seen used is `XmGEO_CENTER`. I suspect that the other two might be used by RowColumn, and possibly Form. What happens if the `fill_mode` is `XmGEO_CENTER` is that extra fill space is distributed between the children in a row; if not `XmGEO_CENTER`, the children are resized proportionally.

`unsigned char fit_mode;`
: One of `XmGEO_PROPORTIONAL`, `XmGEO_AVERAGING`, or `XmGEO_WRAP`. `XmGEO_PROPORTIONAL` means layout the widgets in this row in proportion to the individual sizes of each widget. `XmGEO_AVERAGING` means layout the children based on the average dimensions of all children. `XmGEO_WRAP` means if we can't fit the children on one line, wrap them around to what is effectively another row. You can see this behavior when you resize a M*TIF Dialog to be taller and narrower than it wants to be.

`Boolean sticky_end;`
: Indicates that the last box in the row should be as close to the right margin as possible.

`Dimension space_above;`
: Indicates the amount of space that should be left above this row. if the top row's `space_above` is less than the requested margin, the margin is used.

`Dimension space_end;`
: Indicates the amount of space that should be left at the ends of the row.

`Dimension space_between;`
: Indicates how much space should be between the widgets in a row.

`Dimension max_box_height;`
: Indicates the hight of the largest box in the row.

`Dimension boxes_width;`
: Indicates the cumulative width of all the widgets in a row.

`Dimension fill_width;`
: Indicates the cumulative fill space in a row, both between widgets and at the row end.

`Dimension box_count;`
: The number of boxes in the row.

In general, the user is only interested in fields up to (and including) `space_between`. The remaining fields are used during the calculations.

### 5.3.3 The KidGeometryRec

The final structure is the most important one: the `XmKidGeometry` structure. This structure contains the geometry for a child, and provides a storage place during the layout calculations for that geometry while the algorithms proceed.

```
typedef struct _XmKidGeometryRec {
```

```
    Widget kid;
    XtWidgetGeometry box;
} XmKidGeometryRec, *XmKidGeometry;
```

## 5.4 The GeoUtils Functions

Let's examine the published interface, as these are really the functions that must be understood if you want to understand BulletinBoard, and its GeoUtils using subclasses. By the way, in M*TIF, certain GeoUtils functions are also used by RowColumn. I'll indicate these as each function is discussed (well, at least the ones I know about).

### 5.4.1 The Allocation, Initialization, and Deallocation Functions

First, let's talk about the functions used when you want to allocate a GeoMatrix (i.e., the widget method known as `geo_matrix_create()`). In that method, you have to compute the number of rows of children that you are going to layout, and the number of children (total) involved (more on this when we actually examine a subclass).

The first function, which is actually called from subclass code, takes the information you've gathered about your children, and allocates the GeoMatrix, the `XmGeoMajorLayout`(s) structures, the `XmKidGeometry`(s) structures, and mallocs an extra "`extSize`" bytes for any extension that will be used. It returns a pointer to the allocated structure. Note that it doesn't fill in any information in the matrix – it just allocates it.

```
XmGeoMatrix _XmGeoMatrixAlloc(unsigned int numRows,
                              unsigned int numBoxes,
                              unsigned int extSize);
```

The next function verifies that the child being examined is valid, and sets up the `XmKidGeometry` structure to point at this kid.

```
Boolean _XmGeoSetupKid(XmKidGeometry geo, Widget kidWid);
```

For those of you interested in writing subclass widgets, those two functions are all you really need to know (well, except for knowing how to use them, of course). The rest of the functions are either internal, or buried within the BulletinBoard class.

Finally, when geometry management is complete, the following function deallocates the matrix.

```
void _XmGeoMatrixFree(XmGeoMatrix geo_spec);
```

### 5.4.2 Layout Management Functions

Layout management is essentially a five phase process:

- We ask our children how they want to look.
- We figure out how that would make us look.
- We ask our parent if we can look that way.
- We take how our parent says we can look, and recompute how we want to look.
- We apply the recomputed look to our children.

#### 5.4.2.1 Querying the Children

This first function queries all the manager's children for their geometry. The parameter `geoType` can actually be several different values, but in practice I've never seen the GeoUtils use anything other than `XmGET_PREFERRED_SIZE`.

```
void _XmGeoMatrixGet(XmGeoMatrix geoSpec, int geoType);
```

The pseudo code for this function is as follows:

```
_XmGeoMatrixGet()
{
    while (rows remaining) {
        if (end of row)
            advance to next row
        else
            _XmGeoLoadValues(kid);
}
```

`_XmGeoMatrixGet()` uses a lower level function to ask children their prefered goemetry: `_Xm-GeoLoadValues()`. The behavior of the function is slightly different when the child is the instigator of a geometry management conversation.

```
void _XmGeoLoadValues(Widget wid, int geoType, Widget instigator,
                      XtWidgetGeometry *request,
                      XtWidgetGeometry *geoResult);
```

Testing indicates that this function is used by the RowColumn in M*TIF.

#### 5.4.2.2 Computing the Desired Size

The next function is the real workhorse in layout computation. Basically, it takes the information that was recorded from the previous step and determines how that would make us look. It uses values in the `GeoMatrix`, the `MajorLayoutRec`, and the `KidGeometryRec` to determine this. It uses this information, in combination with the input parameters, to determine how the total composite should look.

```
void
_XmGeoArrangeBoxes(XmGeoMatrix geoSpec, Position x, Position y,
                   Dimension *pW, Dimension *pH);
```

The pseudo code for this function is as follows:

```
_XmGeoArrangeBoxes()
{
    if (user specified an arrange procedure) {
        call user's arrange
        return
    }
    _XmGeoAdjustBoxes()
    _XmGeoGetDimensions()

    adjust the overall layout based on the input parameters

    while (rows remaining)
        _XmGeoArrangeList(row);
    if (height needs adjusting) {
        if (user allows stretching)
            _XmGeoStretchVertical()
        else
            _XmGeoFillVertical()
    }
}
```

`_XmGeoArrangeBoxes()` calls this next function to determine the overall layout. In the current implementation, `_XmGeoAdjustBoxes()` loops through the rows in the composite, figuring out how each row would look.

```
void _XmGeoAdjustBoxes(XmGeoMatrix geoSpec);
```

The pseudo code for this function is as follows:

```
_XmGeoAdjustBoxes()
{
    while (rows remaining) {
        if (children in row should be even width)
            _XmGeoBoxesSameWidth();
        if (children in row should be even height)
            _XmGeoBoxesSameHeight();
        if (children in row should have the same border)
            adjust the border
    }
}
```

If the relevant flags are set in the `MajorLayoutRec`, `_XmGeoAdjustBoxes()` invokes the following two functions:

```
Dimension
_XmGeoBoxesSameWidth(XmKidGeometry rowPtr, Dimension width);
Dimension
_XmGeoBoxesSameHeight(XmKidGeometry rowPtr, Dimension height);
```

Next, `_XmGeoArrangeBoxes()` calls this next function to compute the total picture of the desired geometry. This function takes the overal results computed above, and adjusts values in the Matrix and Layout data structures.

```
void
_XmGeoGetDimensions(XmGeoMatrix geoSpec);
```

#### 5.4.2.3 Computing the Layout

This next function is responsible for actually laying out each row. It is in this function that things like the `fit_mode` and `fill_mode` in the `Layout` structure are evaluated.

```
Position
_XmGeoArrangeList(XmKidGeometry boxes, XmGeoRowLayout layout,
                  Position x, Position y,
                  Dimension width, Dimension margin);
```

The pseudo code for this function is as follows:

```
_XmGeoArrangeList()
{
    figure out the width of our children
    figure out the "fill" space wanted
    figure out the amount of adjusting necessary
    figure out the starting height of this row

    if (things aren't going to fit, and layout fit_mode is XmGEO_WRAP) {
        _XmGeoLayoutWrap()
        return
    }
    else if (things aren't going to fit) {
        if (fit_mode is Xm_GEO_AVERAGING)
            FitBoxesAveraging()
        else
            FitBoxesProportional()
    }
    else if (the wanted width is wider than necessary) {
        if (fill_mode is XmGEO_CENTER)
            _XmGeoCalcFill()
        else
            FitBoxesProporitional()
    }
    _XmGeoLayoutSimple()
}
```

Finally, after the rows have been laid out, the y offsets or the widget heights in each row may need adjusting, based on the actual height of the widget, and the value of `stretch_height`. `_XmGeoArrangeBoxes()` takes care of that with the following two functions. The first stretches the rows to fit; the second inserts filler space.

```
Dimension
_XmGeoStretchVertical(XmGeoMatrix geoSpec, Dimension height, Dimension maxh);
Dimension
_XmGeoFillVertical(XmGeoMatrix geoSpec, Dimension height, Dimension maxh);
```

I'll stop at this level. If you want to know more, you'll need to delve into the code in `GeoUtils.c`. What I've given should be enough for you to find your way around.

#### 5.4.2.4 Applying the Changes

If after all the above computations have happened, and our parent has agreed to our resize request, we call the following function:

```
void _XmGeoMatrixSet(XmGeoMatrix geoSpec);
```

The pseudo code for this is as follows:

```
_XmGeoMatrixSet()
{
    for (each row) {
        for (each child in row)
            _XmSetKidGeo()
    }
}
```

The lower level function `_XmSetKidGeo()` usually calls `_XmConfigureObject()`. The behavior is slightly different during geometry management conversations.

```
void _XmSetKidGeo(XmKidGeometry kg, Widget instigator);
```

### 5.4.3 The Method Functions

The method functions basically implement most of the behavior for certain Xt required methods; the GeoUtils provide default implementations for `set_values()`, `query_geometry()`, and `geometry_manager()`.

The `set_values()` case (really, the implementation of `_XmBulletinBoardSizeUpdate()` – see the section on BulletinBoard `set_values()`) is handled by `_XmHandleSizeUpdate()`. This function is much like the `change_managed()` method in BulletinBoard (see the relevant section for details).

```
void
_XmHandleSizeUpdate(Widget wid, unsigned char policy,
                    XmGeoCreateProc createMatrix);
```

The `query_geometry()` method is handled by the following function. Essentially, this function implements the geometry calculation without doing the layout common to the other methods.

```
XtGeometryResult
_XmHandleQueryGeometry(Widget wid,
                       XtWidgetGeometry *intended,
                       XtWidgetGeometry *desired,
                       unsigned char policy,
                       XmGeoCreateProc createMatrix);
```

The next function handles the `geometry_manager()` method. It is truly a nasty function, and was very difficult to figure out. This is the only place in the entire GeoUtils functionality where the cache is used (and really, it's the only place where it needs to be used). There are some pretty good reasons for this – manager children tend to loop around `XtMakeResizeRequest()`, or `XtMakeGeometryRequest()`, until their parent says yes or no. By using a cache, you can eliminate at least one iteration of the negotiation (which is relatively expensive). I don't really think most readers of this document are interested in the gory details. If you are, reading through the code should give you the necessary information.

```
XtGeometryResult
_XmHandleGeometryManager(Widget wid, Widget instigator,
                         XtWidgetGeometry *desired,
                         XtWidgetGeometry *allowed,
                         unsigned char policy,
                         XmGeoMatrix *cachePtr,
                         XmGeoCreateProc createMatrix);
```

These next two functions do most of the default behavior for the BulletinBoard, when the widget class does not use the GeoUtils (unless, of course, the subclass overrides them).

This first function is invoked when a parent queries a BulletinBoard widget for its prefered size.

```
XtGeometryResult
_XmGMHandleQueryGeometry(Widget w,
                         XtWidgetGeometry *proposed, XtWidgetGeometry *answer,
                         Dimension margin_width, Dimension margin_height,
                         unsigned char resize_policy)
```

The second function is invoked when a child queries a BulletinBoard parent for a resize.

```
XtGeometryResult
_XmGMHandleGeometryManager(Widget w, Widget instigator,
                           XtWidgetGeometry *desired,
                           XtWidgetGeometry *allowed,
                           Dimension margin_width, Dimension margin_height,
                           unsigned char resize_policy, Boolean allow_overlap)
```

### 5.4.4 Miscellaneous Functions

This function determines if two widget geometries are identical.

```
Boolean
_XmGeometryEqual(Widget wid, XtWidgetGeometry *geoA,
                             XtWidgetGeometry *geoB);
```

The next function checks the desired geometry `desired` of the widget `wid` against the parent's proposal in `response`. It returns `True` only if the position (x, y), as well as the width and height, and finally the border width are equal in the desired and proposed geometry. If any one of these five geometry characteristics is of no concern (the corresponding bit in `response->request_mode` is unset) then it will be ignored.

```
Boolean
_XmGeoReplyYes(Widget wid, XtWidgetGeometry *desired,
                           XtWidgetGeometry *response);
```

This function asks the parent of the widget `w` for a new desired geometry `geom` of `w`. If the parent answers `XtGeometryAlmost` then we ask him a second time with the proposed geometry, so he can accept and set the new geometry. If the parent refuses in any way then we'll inform the user of our parent's impudent habbits (because he doesn't conform to the geometry negotiation protocol, see O'Reilly, Vol. 5, pp. 264 for details about `XtMakeGeometryRequest`). This function is at least only a convenience function but according to informed sources, M*TIF never issues a bare `XtMakeGeometryRequest` but uses always `_XmMakeGeometryRequest`.

```
XtGeometryResult
_XmMakeGeometryRequest(Widget w, XtWidgetGeometry *geom);
```

The next function erases the background at both the old and the new position of a rectangle object. Whereas the old position must be explicitely specified in `old` the new position is determined from `w`. Redrawing events will be triggered (and queued) for the affected areas.

```
void
_XmGeoClearRectObjAreas(RectObj r, XWindowChanges *old);
```

The next one reappears as `XmeReplyToQueryGeometry` in M*TIF 2.0. This function is a shortcut for simple `query_geometry` methods which are interested only in their width and height but neither their position nor border width. To use it, first compute your desired width and height in your widget's `query_geometry` method and then return the result from the call to `_XmGMReplyToQueryGeometry`.

```
void
_XmGMReplyToQueryGeometry(void);
```

These next two functions are "fixup" functions. They are invoked by _XmGeoMatrixSet(), to override the generic geometry computation with specific behavior. _XmMenuBarFix() forces a menu bar to be the full width of its composite parent. _XmSeparatorFix() does the same for separators.

```
void
_XmMenuBarFix(XmGeoMatrix geoSpec, int action,
              XmGeoMajorLayout layoutPtr, XmKidGeometry rowPtr);
void
_XmSeparatorFix(XmGeoMatrix geoSpec, int action,
                XmGeoMajorLayout layoutPtr, XmKidGeometry rowPtr);
```

### 5.4.5 BulletinBoard Helper Functions

The next several functions implement bits of BulletinBoard behavior.

This next function ensures that a BulletinBoard child is constrained within the margins of the BulletinBoard.

```
void
_XmGMEnforceMargin(Widget w,
                   Dimension margin_width, Dimension margin_height,
                   Boolean useSetValues)
```

The next function implements the XmNallowOverlap behavior (or rather, if XmNallowOverlap is False, makes sure that children do not overlap).

```
Boolean
_XmGMOverlap(Widget w, Widget instigator,
             Position x, Position y, Dimension width, Dimension height)
```

The next function computes the desired size of a BulletinBoard.

```
void
_XmGMCalcSize(Widget w, Dimension margin_w, Dimension margin_h,
              Dimension *retw, Dimension *reth)
```

The next function performs the BulletinBoard layout behavior.

```
void
_XmGMDoLayout(Widget w, Dimension margin_w, Dimension margin_h,
              unsigned char resize_policy, short adjust)
```

### 5.4.6 RowColumn Specific Functions

The following functions are specific to RowColumn functionality, but reside in the GeoUtils implementation.

I don't know what this first function does, but it looks suspiciously like a `GeoMatrix` allocation function, specialized for `RCKidGeometry`.

```
XmKidGeometry
_XmGetKidGeo(Widget wid, Widget instigator,
             XtWidgetGeometry *request,
             int uniform_border, Dimension border,
             int uniform_width_margins,
             int uniform_height_margins,
             Widget help, int geo_type);
```

There is an undocumented function, `XmRCGetKidGeo()`, that I believe is similar to `_XmGeoMatrixGet()`. In LESSTIF, I believe this is implemented as `initialize_boxes()` in `RowColumn.c`.

That function calls this next function, as well as `_XmGeoLoadValues()`.

```
int _XmGeoCount_kids(CompositeWidget c);
```

Anybody out there who knows that this function does?

## 5.5 How to Build a Subclass Using the GeoUtils

At this point, we've come full circle. Now that you know something about how the GeoUtils work, let's examine how a subclass can use them. I'll now talk about the `XmTrivial` widget class. It doesn't implement anything more than an even layout of the button children as if they were action buttons in a dialog. The order of the buttons in the layout is the same as the creation order. In order to keep it simple, there are no new Xt or synthetic resources beyond those covered by BulletinBoard. If you are thinking about using the GeoUtils, the `Trivial` class makes a pretty good template. The tested implementation lives in `$(LESSTIF_ROOT)/testXm/geometry`.

### 5.5.1 The Header Files

There really isn't much to say about the header files. They are pretty much standard headers for a widget implementation (same procedure as every widget...). Here's the public header (from the file `Trivial.h`):

```
#ifndef TRIVIAL_H
```

```
#define TRIVIAL_H

#include <Xm/Xm.h>
#include <Xm/BulletinB.h>

extern WidgetClass xmTrivialWidgetClass;

typedef struct _XmTrivialRec *XmTrivialWidget;
typedef struct _XmTrivialConstraintRec *XmTrivialConstraint;

#ifndef XmIsTrivial
#define XmIsTrivial(a) (XtIsSubclass(a, xmTrivialWidgetClass))
#endif

Widget XmCreateCreateTrivial(Widget _p, char *_n, ArgList _a,
                             Cardinal _narg);

#endif
```

Now, the private header (`TrivialP.h`):

```
#ifndef TRIVIAL_P_H
#define TRIVIAL_P_H

#include <Xm/XmP.h>
#include <Xm/BulletinBP.h>
#include "Trivial.h"

typedef struct _XmTrivialClassPart {
        int duh;
} XmTrivialClassPart;

typedef struct _XmSmartMessageBoxClassRec {
        CoreClassPart            core_class;
        CompositeClassPart       composite_class;
        ConstraintClassPart      constraint_class;
        XmManagerClassPart       manager_class;
        XmBulletinBoardClassPart bulletin_board_class;
        XmTrivialClassPart       trivial_class;
} XmTrivialClassRec, *XmTrivialWidgetClass;

typedef struct _XmTrivialPart {
        int gaah;
} XmTrivialPart;

typedef struct _XmTrivialRec {
        CorePart            core;
        CompositePart       composite;
        ConstraintPart      constraint;
        XmManagerPart       manager;
        XmBulletinBoardPart bulletin_board;
        XmTrivialPart       trivial;
} XmTrivialRec, *XmTrivialPtr;

#endif
```

### 5.5.2 The Implementation

In this section I'll describe the various sections in `Trivial.c` that are important to the subclass. The new sections are printed in a **`bold typewriter`** font. Important parts of the new sections are typeset in an ***`italic bold typewriter`*** font.

```
#include <LTconfig.h>
#include <Xm/XmP.h>
#include <Xm/XmI.h>
#include <Xm/BulletinBP.h>
#include <Xm/PushBP.h>
#include <Xm/PushBGP.h>
#include <Xm/DebugUtil.h>
#include "TrivialP.h"

/*
 * Forward Declarations
 */
static void class_initialize();
static void class_part_initialize(WidgetClass class);
static void initialize(Widget request, Widget new,
                       ArgList args, Cardinal *num_args);
static void destroy(Widget w);
static Boolean set_values(Widget current, Widget request, Widget new,
                          ArgList args, Cardinal *num_args);

XmGeoMatrix trivial_matrix_create(Widget _w, Widget _from,
                                  XtWidgetGeometry *_pref);
Boolean trivial_NoGeoRequest(XmGeoMatrix _geoSpec);

static XmBaseClassExtRec _XmTrivialCoreClassExtRec = {
    /* next_extension               */ NULL,
    /* record_type                  */ NULLQUARK,
    /* version                      */ XmBaseClassExtVersion,
    /* size                         */ sizeof(XmBaseClassExtRec),
    /* initialize_prehook           */ NULL,
    /* set_values_prehook           */ NULL,
    /* initialize_posthook          */ NULL,
    /* set_values_posthook          */ NULL,
    /* secondary_object_class       */ NULL,
    /* secondary_object_create      */ NULL,
    /* get_secondary_resources      */ NULL,
    /* fast_subclass                */ { 0 },
    /* get_values_prehook           */ NULL,
    /* get_values_posthook          */ NULL,
    /* class_part_init_prehook      */ NULL,
    /* class_part_init_posthook     */ NULL,
    /* ext_resources                */ NULL,
    /* compiled_ext_resources       */ NULL,
    /* num_ext_resources            */ 0,
    /* use_sub_resources            */ FALSE,
    /* widget_navigable             */ NULL,
    /* focus_change                 */ NULL,
    /* wrapper_data                 */ NULL
};

static XmManagerClassExtRec _XmTrivialMClassExtRec = {
```

```
    /* next_extension               */ NULL,
    /* record_type                  */ NULLQUARK,
    /* version                      */ XmManagerClassExtVersion,
    /* record_size                  */ sizeof(XmManagerClassExtRec),
    /* traversal_children           */ NULL /* FIXME */
};

XmTrivialClassRec xmTrivialClassRec = {
    /* Core class part */
    {
    /* superclass                   */ (WidgetClass)
                                          &xmBulletinBoardClassRec,
    /* class_name                   */ "XmTrivial",
    /* widget_size                  */ sizeof(XmBulletinBoardRec),
    /* class_initialize             */ class_initialize,
    /* class_part_initialize        */ class_part_initialize,
    /* class_inited                 */ FALSE,
    /* initialize                   */ initialize,
    /* initialize_hook              */ NULL,
    /* realize                      */ XtInheritRealize,
    /* actions                      */ NULL,
    /* num_actions                  */ 0,
    /* resources                    */ NULL,
    /* num_resources                */ 0,
    /* xrm_class                    */ NULLQUARK,
    /* compress_motion              */ TRUE,
    /* compress_exposure            */ XtExposeCompressMaximal,
    /* compress_enterleave          */ TRUE,
    /* visible_interest             */ FALSE,
    /* destroy                      */ destroy,
    /* resize                       */ XtInheritResize,
    /* expose                       */ XtInheritExpose,
    /* set_values                   */ set_values,
    /* set_values_hook              */ NULL,
    /* set_values_almost            */ XtInheritSetValuesAlmost,
    /* get_values_hook              */ NULL,
    /* accept_focus                 */ NULL,
    /* version                      */ XtVersion,
    /* callback offsets             */ NULL,
    /* tm_table                     */ NULL,
    /* query_geometry               */ XtInheritQueryGeometry,
    /* display_accelerator          */ NULL,
    /* extension                    */ (XtPointer)
                                          &_XmTrivialCoreClassExtRec
    },
    /* Composite class part */
    {
    /* geometry_manager             */ XtInheritGeometryManager,
    /* change_managed               */ XtInheritChangeManaged,
    /* insert_child                 */ XtInheritInsertChild,
    /* delete_child                 */ XtInheritDeleteChild,
    /* extension                    */ NULL
    },
    /* Constraint class part */
    {
    /* subresources                 */ NULL,
    /* subresource_count            */ 0,
    /* constraint_size              */ 0,
    /* initialize                   */ NULL,
```

```
    /* destroy                        */ NULL,
    /* set_values                     */ NULL,
    /* extension                      */ NULL
    },
    /* XmManager class part */
    {
    /* translations                   */ XtInheritTranslations,
    /* syn_resources                  */ NULL,
    /* num_syn_resources              */ 0,
    /* syn_constraint_resources       */ NULL,
    /* num_syn_constraint_resources */ 0,
    /* parent_process                 */ XmInheritParentProcess,
    /* extension                      */ (XtPointer)
                                           &_XmTrivialMClassExtRec
    },
    /* XmBulletinBoard Area part */
    {
    /* always_install_accelerators  */ False,
    /* geo_matrix_create            */ trivial_matrix_create,
    /* focus_moved_proc             */ XmInheritFocusMovedProc,
    /* extension                    */ NULL

    },
    /* XmTrivial Class Part */
    {
    /* extension                    */ 0
    }
};

WidgetClass xmTrivialWidgetClass = (WidgetClass)&xmTrivialClassRec;

static void
class_initialize()
{
    _XmTrivialCoreClassExtRec.record_type = XmQmotif;
}

static void
class_part_initialize(WidgetClass widget_class)
{
}

static void
initialize(Widget request,
           Widget new,
           ArgList args,
           Cardinal *num_args)
{
}

static void
destroy(Widget w)
{
}

static Boolean
set_values(Widget old,
           Widget request,
           Widget new,
```

```
            ArgList args,
            Cardinal *num_args)
{
    Boolean refresh_needed = False;

    BB_InSetValues(new) = True;

    /* do any class specific stuff */

    BB_InSetValues(new) = False;

    if (refresh_needed && (XtClass(new) == xmTrivialWidgetClass))
    {
        _XmBulletinBoardSizeUpdate(new);
        return False;
    }
    return refresh_needed;
}

XmGeoMatrix
trivial_matrix_create(Widget _w, Widget _from, XtWidgetGeometry *_pref)
{
    XmGeoMatrix geoSpec;
    register XmGeoRowLayout layoutPtr;
    register XmKidGeometry boxPtr;
    Cardinal numKids;
    int i, nrows;
    Widget child;

    numKids = MGR_NumChildren(_w);

    /* compute the number of rows you want here. */
    nrows = 1; /* Trivial only has one */

    geoSpec = _XmGeoMatrixAlloc(nrows, numKids, 0);
    geoSpec->composite = (Widget)_w;
    geoSpec->instigator = (Widget)_from;
    if (_pref)
        geoSpec->instig_request = *_pref;
    geoSpec->margin_w = BB_MarginWidth(_w) + MGR_ShadowThickness(_w);
    geoSpec->margin_h = BB_MarginHeight(_w) + MGR_ShadowThickness(_w);
    geoSpec->no_geo_request = trivial_NoGeoRequest;

    layoutPtr = &(geoSpec->layouts->row);
    boxPtr = geoSpec->boxes;

    /* row 1 */
    layoutPtr->fill_mode = XmGEO_CENTER;
    layoutPtr->fit_mode = XmGEO_WRAP;
    layoutPtr->even_width = 1;
    layoutPtr->even_height = 1;
    layoutPtr->space_above = BB_MarginHeight(_w);
    for (i = 0; i < numKids; i++) {
        child = MGR_Children(_w)[i];
        if ((XmIsPushButton(child) || XmIsPushButtonGadget(child)) &&
            XtIsManaged(child) && _XmGeoSetupKid(boxPtr, child))
        {
            boxPtr++;
        }
```

```
    }
    layoutPtr++;
    /* end marker */
    layoutPtr->space_above = 0;
    layoutPtr->end = TRUE;

    return(geoSpec);
}

Boolean
trivial_NoGeoRequest(XmGeoMatrix geo)
{
    if (BB_InSetValues(geo->composite) &&
        XtClass(geo->composite) == xmTrivialWidgetClass)
        return TRUE;

    return FALSE;
}
```

Not bad, only around 350 lines of code. This is about the mininum you can get away with if you write a manager widget anyway. But now let us go straight into the details.

#### 5.5.2.1 Extra Prototypes

You'll need to provide two extra prototypes for a GeoUtils subclass - one for the `geo_matrix_create()` method, and one for the `no_geo_request()` method. These should match the types specified in `XmP.h`. From `Trivial.c`:

```
XmGeoMatrix trivial_matrix_create(Widget _w, Widget _from,
                                  XtWidgetGeometry *_pref);
Boolean trivial_NoGeoRequest(XmGeoMatrix _geoSpec);
```

#### 5.5.2.2 The Class Structure

The first thing to know is how to type `XtInherit`. Unless you really know what you are doing, and want to override specific behaviors, you should definitely specify `XtInherit` in the class structure of your subclass for the following methods:

- `realize()`,
- `resize()`,
- `expose()`,
- `query_geometry()`,
- `geometry_manager()`,
- `change_managed()`.

Unless you are implementing a fairly trivial widget (such as `XmTrivial`), you'll probably have to provide your own `set_values()` method. That's okay, just make sure you follow the rules outlined in the BulletinBoard section above.

#### 5.5.2.3 The set_values() Method

In any interesting widget, the `set_values()` method will probably do something (but it doesn't in `Trivial.c`). The code below can be considered boilerplate; you should probably base a subclass's `set_values()` method on this code. Note especially the region reserved for setting class specific instance variables.

```
static Boolean
set_values(Widget old,
           Widget request,
           Widget new,
           ArgList args,
           Cardinal *num_args)
{
    Boolean refresh_needed = False;

    BB_InSetValues(new) = True;
    /* do any class specific stuff HERE */
    BB_InSetValues(new) = False;

    if (refresh_needed && (XtClass(new) == xmTrivialWidgetClass))
    {
        _XmBulletinBoardSizeUpdate(new);
        return False;
    }
    return refresh_needed;
}
```

#### 5.5.2.4 The NoGeoRequest Method

This method actually doesn't get placed in the class structure, but rather in the GeoMatrix during its creation. Again, the implementation in `Trivial.c` is boilerplate; the only thing a subclass needs to do is change the tested widget class:

```
Boolean
trivial_NoGeoRequest(XmGeoMatrix geo)
{
    if (BB_InSetValues(geo->composite) &&
        XtClass(geo->composite) == xmTrivialWidgetClass)
        return TRUE;

    return FALSE;
}
```

#### 5.5.2.5 The GeoMatrixCreate Method

Now we get to the interesting part of the implementation. The `geo_matrix_create()` method in `Trivial.c` is *not* boilerplate, but it does show you what you need to do (well, actually, one small portion is boilerplate). Instead of repeating the code section here you can look up the method in the listing on the preceding pages – but take note that the method in `Trivial.c` is called `trivial_matrix_create()`.

Note that the function has essentially three sections. In the first section, you need to loop through your children (or evaluate instance variables, as is done in SelectionBox), deciding on how many rows of children that need to be controlled. Basically, what you are doing is evaluating how many `MajorLayout` structures you are going to need. You can also choose to count the number of managed children you have (this may or may not be the same as the number of children you have); this is optional, as the wasted space is not very large, and it eventually gets deallocated anyway.

In the second section, we have a small piece of boilerplate: it is very important to duplicate this code exactly. While the `_pref` and `_from` fields are often `NULL`, they are *not* when this method is called from `_XmHandleGeometryManager()`. Make sure you copy this right.

```
    geoSpec = _XmGeoMatrixAlloc(nrows, numKids, 0);
    geoSpec->composite = (Widget)_w;
    geoSpec->instigator = (Widget)_from;
    if (_pref)
        geoSpec->instig_request = *_pref;
    geoSpec->margin_w = BB_MarginWidth(_w) + MGR_ShadowThickness(_w);
    geoSpec->margin_h = BB_MarginHeight(_w) + MGR_ShadowThickness(_w);
    geoSpec->no_geo_request = trivial_NoGeoRequest;
```

You can be a little creative when you calculate the `margin_w` and `margin_h` variables. Also, make sure that you hook up the `no_geo_request()` method as shown in the last line of the code excerpt above.

The third section of code is basically where the subclass needs to setup the `MajorLayout` structures with the desired information for controlling the layout, and setting the `KidGeometry` structures to point at the widget children that should appear.

`XmTrivial`'s implementation of this method is *very* simplistic. Now for a little more demanding example. The following code is SelectionBox's version – look for the boilerplate above to find the separation between the sections:

```
XmGeoMatrix
_XmSelectionBoxGeoMatrixCreate(Widget _w, Widget _from,
                               XtWidgetGeometry *_pref)
{
    XmGeoMatrix geoSpec;
    register XmGeoRowLayout layoutPtr;
    register XmKidGeometry boxPtr;
    Cardinal numKids;
    Boolean newRow;
    int nrows, i, nextras;
    Widget *extras;

    numKids = MGR_NumChildren(_w);

    nextras = 0;
    extras = NULL;
```

```
for (i = 0; i < numKids; i++)
{
    if (XtIsManaged(MGR_Children(_w)[i]) &&
        MGR_Children(_w)[i] != SB_ListLabel(_w) &&
        (SB_List(_w)
            ? MGR_Children(_w)[i] != XtParent(SB_List(_w))
            : True) &&
        MGR_Children(_w)[i] != SB_SelectionLabel(_w) &&
        MGR_Children(_w)[i] != SB_Text(_w) &&
        MGR_Children(_w)[i] != SB_Separator(_w) &&
        MGR_Children(_w)[i] != SB_OkButton(_w) &&
        MGR_Children(_w)[i] != SB_ApplyButton(_w) &&
        MGR_Children(_w)[i] != SB_HelpButton(_w) &&
        MGR_Children(_w)[i] != BB_CancelButton(_w))
    {
        nextras++;
    }
}

if (nextras)
    extras = (Widget *)XtMalloc(sizeof(Widget) * nextras);

nextras = 0;
for (i = 0; i < numKids; i++)
{
    if (XtIsManaged(MGR_Children(_w)[i]) &&
        MGR_Children(_w)[i] != SB_ListLabel(_w) &&
        (SB_List(_w)
            ? MGR_Children(_w)[i] != XtParent(SB_List(_w))
            : True) &&
        MGR_Children(_w)[i] != SB_SelectionLabel(_w) &&
        MGR_Children(_w)[i] != SB_Text(_w) &&
        MGR_Children(_w)[i] != SB_Separator(_w) &&
        MGR_Children(_w)[i] != SB_OkButton(_w) &&
        MGR_Children(_w)[i] != SB_ApplyButton(_w) &&
        MGR_Children(_w)[i] != SB_HelpButton(_w) &&
        MGR_Children(_w)[i] != BB_CancelButton(_w))
    {
        extras[nextras] = MGR_Children(_w)[i];
        nextras++;
    }
}

nrows = 0;

/* note the starting from one.  The zero'th child is the "work area" */
if (nextras > 0) {
    for (i = 1; i < nextras; i++) {
        if (XmIsMenuBar(extras[i]) && XtIsManaged(extras[i]))
            nrows++;
    }
    if (extras[0] && XtIsManaged(extras[0]))
        nrows++;
}

if (SB_ListLabel(_w) && XtIsManaged(SB_ListLabel(_w)))
    nrows++;

if (SB_List(_w) && XtIsManaged(SB_List(_w)))
```

```
        nrows++;

    if (SB_SelectionLabel(_w) && XtIsManaged(SB_SelectionLabel(_w)))
        nrows++;

    if (SB_Text(_w) && XtIsManaged(SB_Text(_w)))
        nrows++;

    if (SB_Separator(_w) && XtIsManaged(SB_Separator(_w)))
        nrows++;

    if ((BB_CancelButton(_w) && XtIsManaged(BB_CancelButton(_w))) ||
         (SB_OkButton(_w)     && XtIsManaged(SB_OkButton(_w))) ||
         (SB_ApplyButton(_w)  && XtIsManaged(SB_ApplyButton(_w))) ||
         (SB_HelpButton(_w)   && XtIsManaged(SB_HelpButton(_w))))
        nrows++;
    else {
        for (i = i; i < nextras; i++) {
            if (extras[i] && XtIsManaged(extras[i]) &&
                (XmIsPushButton(extras[i]) ||
                 XmIsPushButtonGadget(extras[i])))
            {
                nrows++;
                break;
            }
        }
    }

    geoSpec = _XmGeoMatrixAlloc(nrows, numKids, 0);
    geoSpec->composite = (Widget)_w;
    geoSpec->instigator = (Widget)_from;
    if (_pref)
        geoSpec->instig_request = *_pref;
    geoSpec->margin_w = BB_MarginWidth(_w) + MGR_ShadowThickness(_w);
    geoSpec->margin_h = BB_MarginHeight(_w) + MGR_ShadowThickness(_w);
    geoSpec->no_geo_request = _XmSelectionBoxNoGeoRequest;

    layoutPtr = &(geoSpec->layouts->row);
    boxPtr = geoSpec->boxes;

    for (i = 1; i < nextras; i++) {
        if (XmIsMenuBar(extras[i]) && XtIsManaged(extras[i]))
        {
            layoutPtr->fix_up = _XmMenuBarFix;
            layoutPtr->space_above = 0;
            boxPtr += 2;
            layoutPtr++;
        }
    }

    if (SB_ChildPlacement(_w) == XmPLACE_TOP && nextras &&
        extras[0] && XtIsManaged(extras[0]) &&
        _XmGeoSetupKid(boxPtr, extras[0]))
    {
            layoutPtr->stretch_height = 1;
            layoutPtr->fill_mode = XmGEO_EXPAND;
            layoutPtr->even_width = 1;
            layoutPtr->even_height = 1;
            layoutPtr->space_above = BB_MarginHeight(_w);
```

```
        layoutPtr++;
        boxPtr += 2;
        nrows++;
}

if (SB_DialogType(_w) == XmDIALOG_PROMPT &&
    SB_ChildPlacement(_w) == XmPLACE_ABOVE_SELECTION && nextras &&
    extras[0] && XtIsManaged(extras[0]) &&
    _XmGeoSetupKid(boxPtr, extras[0]))
{
        layoutPtr->stretch_height = 1;
        layoutPtr->fill_mode = XmGEO_EXPAND;
        layoutPtr->even_width = 1;
        layoutPtr->even_height = 1;
        layoutPtr->space_above = BB_MarginHeight(_w);
        layoutPtr++;
        boxPtr += 2;
        nrows++;
}

newRow = False;
if (SB_ListLabel(_w) && XtIsManaged(SB_ListLabel(_w)) &&
    _XmGeoSetupKid(boxPtr, SB_ListLabel(_w)))
{
    layoutPtr->fill_mode = XmGEO_EXPAND;
    layoutPtr->fit_mode = XmGEO_PROPORTIONAL;
    layoutPtr->even_width = 1;
    layoutPtr->even_height = 1;
    layoutPtr->space_above = BB_MarginHeight(_w);
    layoutPtr->space_between = BB_MarginWidth(_w);
    newRow = TRUE;
    boxPtr++;
}

if (newRow)
{
    layoutPtr++;
    boxPtr++;
}

if (SB_DialogType(_w) == XmDIALOG_COMMAND &&
    SB_ChildPlacement(_w) == XmPLACE_ABOVE_SELECTION && nextras &&
    extras[0] && XtIsManaged(extras[0]) &&
    _XmGeoSetupKid(boxPtr, extras[0]))
{
        layoutPtr->stretch_height = 1;
        layoutPtr->fill_mode = XmGEO_EXPAND;
        layoutPtr->even_width = 1;
        layoutPtr->even_height = 1;
        layoutPtr->space_above = BB_MarginHeight(_w);
        layoutPtr++;
        boxPtr += 2;
        nrows++;
}

newRow = FALSE;
if (SB_List(_w) && XtIsManaged(SB_List(_w)) &&
    _XmGeoSetupKid(boxPtr, XtParent(SB_List(_w))))
{
```

```
        layoutPtr->stretch_height = 1;
        layoutPtr->fill_mode = XmGEO_EXPAND;
        layoutPtr->fit_mode = XmGEO_PROPORTIONAL;
        layoutPtr->even_width = 1;
        layoutPtr->even_height = 1;
        layoutPtr->space_above = 0; /* BB_MarginHeight(_w); */
        layoutPtr->space_between = BB_MarginWidth(_w);
        newRow = TRUE;
        boxPtr++;
    }

    if (newRow)
    {
        layoutPtr++;
        boxPtr++;
    }

    if (SB_DialogType(_w) != XmDIALOG_COMMAND &&
        SB_DialogType(_w) != XmDIALOG_PROMPT &&
        SB_ChildPlacement(_w) == XmPLACE_ABOVE_SELECTION && nextras &&
        extras[0] && XtIsManaged(extras[0]) &&
        _XmGeoSetupKid(boxPtr, extras[0]))
    {
            layoutPtr->stretch_height = 1;
            layoutPtr->fill_mode = XmGEO_EXPAND;
            layoutPtr->even_width = 1;
            layoutPtr->even_height = 1;
            layoutPtr->space_above = BB_MarginHeight(_w);
            layoutPtr++;
            boxPtr += 2;
            nrows++;
    }

    if (SB_SelectionLabel(_w) && XtIsManaged(SB_SelectionLabel(_w)) &&
        _XmGeoSetupKid(boxPtr, SB_SelectionLabel(_w)))
    {
        layoutPtr->fill_mode = XmGEO_EXPAND;
        layoutPtr->even_width = 0;
        layoutPtr->even_height = 1;
        layoutPtr->space_above = BB_MarginHeight(_w);
        layoutPtr++;
        boxPtr += 2;
    }

    if (SB_Text(_w) && XtIsManaged(SB_Text(_w)) &&
        _XmGeoSetupKid(boxPtr, SB_Text(_w)))
    {
        layoutPtr->fill_mode = XmGEO_EXPAND;
        layoutPtr->stretch_height = 0;
        layoutPtr->even_height = 1;
        layoutPtr->even_width = 0;
        layoutPtr->space_above = 0; /* BB_MarginHeight(_w); */
        boxPtr += 2;
        layoutPtr++;
    }

    if (SB_ChildPlacement(_w) == XmPLACE_BELOW_SELECTION && nextras &&
        extras[0] && XtIsManaged(extras[0]) &&
        _XmGeoSetupKid(boxPtr, extras[0]))
```

```
{
        layoutPtr->stretch_height = 1;
        layoutPtr->fill_mode = XmGEO_EXPAND;
        layoutPtr->even_width = 1;
        layoutPtr->even_height = 1;
        layoutPtr->space_above = BB_MarginHeight(_w);
        layoutPtr++;
        boxPtr += 2;
        nrows++;
}

if (SB_Separator(_w) && XtIsManaged(SB_Separator(_w)) &&
    _XmGeoSetupKid( boxPtr, SB_Separator(_w)))
{
    layoutPtr->fix_up = _XmSeparatorFix;
    layoutPtr->space_above = BB_MarginHeight(_w);
    boxPtr += 2;
    layoutPtr++;
}

newRow = False;
if (SB_OkButton(_w) && XtIsManaged(SB_OkButton(_w)) &&
    _XmGeoSetupKid(boxPtr++, SB_OkButton(_w))) {
    layoutPtr->fill_mode = XmGEO_CENTER;
    layoutPtr->fit_mode = XmGEO_WRAP;
    layoutPtr->even_width = 1;
    layoutPtr->even_height = 1;
    layoutPtr->space_above = BB_MarginHeight(_w);
    newRow = True;
}
for (i = 1; i < nextras; i++)
{
    if (extras[i] && XtIsManaged(extras[i]) &&
        (XmIsPushButton(extras[i]) || XmIsPushButtonGadget(extras[i])) &&
        _XmGeoSetupKid(boxPtr++, extras[i]))
    {
        layoutPtr->fill_mode = XmGEO_CENTER;
        layoutPtr->fit_mode = XmGEO_WRAP;
        layoutPtr->even_width = 1;
        layoutPtr->even_height = 1;
        layoutPtr->space_above = BB_MarginHeight(_w);
        newRow = True;
    }
}

if (SB_ApplyButton(_w)  && XtIsManaged(SB_ApplyButton(_w)) &&
    _XmGeoSetupKid(boxPtr++, SB_ApplyButton(_w))) {
    layoutPtr->fill_mode = XmGEO_CENTER;
    layoutPtr->fit_mode = XmGEO_WRAP;
    layoutPtr->even_width = 1;
    layoutPtr->even_height = 1;
    layoutPtr->space_above = BB_MarginHeight(_w);
    newRow = True;
}
if (BB_CancelButton(_w) && XtIsManaged(BB_CancelButton(_w)) &&
    _XmGeoSetupKid(boxPtr++, BB_CancelButton(_w))) {
    layoutPtr->fill_mode = XmGEO_CENTER;
    layoutPtr->fit_mode = XmGEO_WRAP;
    layoutPtr->even_width = 1;
```

```
        layoutPtr->even_height = 1;
        layoutPtr->space_above = BB_MarginHeight(_w);
        newRow = True;
    }
    if (SB_HelpButton(_w)   && XtIsManaged(SB_HelpButton(_w)) &&
        _XmGeoSetupKid(boxPtr++, SB_HelpButton(_w))) {
        layoutPtr->fill_mode = XmGEO_CENTER;
        layoutPtr->fit_mode = XmGEO_WRAP;
        layoutPtr->even_width = 1;
        layoutPtr->even_height = 1;
        layoutPtr->space_above = BB_MarginHeight(_w);
        newRow = True;
    }

    if (newRow)
    {
        layoutPtr++;
        boxPtr++;
    }

    layoutPtr->space_above = 0; /* BB_MarginHeight(_w); */
    layoutPtr->end = TRUE;
    if (nextras)
        XtFree((char *)extras);
    return(geoSpec);
}
```

While it may look scary, once you understand what it is doing, it really isn't. You can see the advantage of using the GeoUtils in SelectionBox: other than the code above, there really isn't any trace of geometry management in SelectionBox – it's all taken care of automagically.

Another point to note is SelectionBox's `no_geo_request()` method – it's slightly different, as the Command widget class doesn't even *have* a `geo_matrix_create()` method – instead, it inherits SelectionBox's.

```
Boolean
_XmSelectionBoxNoGeoRequest(XmGeoMatrix _geoSpec)
{
    if (BB_InSetValues(_geoSpec->composite) &&
        (XtClass(_geoSpec->composite) == xmSelectionBoxWidgetClass ||
         XtClass(_geoSpec->composite) == xmCommandWidgetClass))
        return TRUE;
    return FALSE;
}
```

## 5.6 Conclusion and Credits

Please keep in mind when reading this document that I'm still discovering new things in the GeoUtils, and I may not be accurate in some places. I'd dearly like feedback from those of you who really know the M*TIF implementation to point out where I'm wrong.

I'd like to thank John Cwikla (again), for providing sample code about how to subclass using

the GeoUtils; Chris, for starting this project in the first place; Danny, for motivating me to write this chapter (thin as it is) – I guess it really *is* a pain to have significant portions of your widgets based on somebody else's undocumented code (I think there may be two or three informational comments in GeoUtils); and the rest of the core team (Rob, Peter, perhaps a few more) for helping out; the team as a whole, for putting up with my mess (what? me opinionated? Nah).

# 6

# Drag and Drop

Harald Albrecht
Mitch Miers

## 6.1 Introduction

The ”Drag & Drop” mechanism is a metaphor for data transfer, which allows the user to pick up an object with the mouse pointer, and move it to another location (which can be even in another application) and drop it there. This metaphor is the same no matter what kind of data is transfered. In most cases, the “Drag & Drop” gesture results in data being moved or copied to the new location, but it can also invoke an action – like printing a file when dropping the icon of a file on a printer icon, or de-installing some so-called “operating system” when dropping its flying logo into the trash bin.

Although the term “Drag & Drop” suggests that both the drag and the drop operation are inseparable, in fact the *drop* operation is independent of the *drag* operation. To transfer data, only the drop operation is needed. The drag operation is just there to make the metaphor work better, as an application can provide accurate visual feedback about the state of the operation. As a consequence of this, we’ll discuss the drag protocol and the drop protocol in separate sections.

As you can easily imagine, a good part of the whole protocol mess is undocumented. Fortunately, Daniel Dardailler’s documentation and initial implementation example of the dynamic drag protocol was invaluable help for understanding the drag and drop protocol. Unfortunately, his documentation does only cover the *dynamic* mode of operation, which is now standardized with CDE 2. The other mode, *preregister*, is still undocumented, although it is the default mode for drag and drop operations in the M*TIF toolkit (one more own goal for the CSF).

## 6.2 Protocol Basics

The top level window where the drag and drop operation starts is called the “source window”. The client who owns this window is called the initiator client, or short “initiator”. The drop operation then takes place in the “destination window” and the client who owns that window is called the “receiver”. The initiator and the receiver of the drag and drop operation can be the same client – but this need not necessarily be always the case. Besides this, even the source and the destination window may be the same.

From the user’s point of view there are no such things like windows recognizable. The only “objects” the user deals with are the “drop sites”, which appear to her/him as entities supplying or receiving information. On the technical side, a drop site may be drawn into its own window or may be drawn as a part of a larger window containing other objects too. The protocol does not put any restrictions on how drop sites are organized within windows. To avoid a possible upcoming confusion, the protocol only knows of and works with the top level windows. Remember, according to the ICCCM 2.0, top level windows are distinguishable from ordinary windows by their `WM_STATE` properties.

Okay, after you’ve now learned some new “drag and drop” buzz words, you’re now ready to dive into the world of bitfields and status codes. Afterwards, you’ll meet the various drag and drop messages which make up the protocol.

### 6.2.1 Drag Operation Modes

Both the drag and the drop protocol are build around X client messages which are send forth and back between the initiator and receiver of the drag and drop operation. Such messages are emitted during the start, cancelation or end of a drag and drop operation, or as the user moves the mouse pointer around the screen and the pointer enters and leaves valid drop sites. This way, the initiator client and the potential receiver client(s) can provide to the user some visual feedback about the current state of the drag operation. Such a visual feedback could be highlighting a drop site so the user notices the existence of a valid drop site. There are two sets of visual feedbacks: the "drag-over visuals" of the initiator and the "drag-under" visuals of the receiver(s).

The drag operation as well as the handling of the drag-under visuals are different depending on the operation mode of the drag:

- When both the initiator and receiver have agreed to use the *dynamic* mode they exchange protocol messages dealing (for example) with entering and leaving drop sites *during* the drag operation. The advantages of this mode are that the X server is not grabbed and can still respond to other event sources, and the receiver can decide what data to accept on-the-fly. On the other side, the dynamic operation mode requires more overhead on behalf of the application and the network.
- The *preregister* mode is the other possible mode of operation (the appropriate protocol is still undocumented). In this mode, the M*TIF toolkit handles on behalf of the initiator client the complete processing of the drag-under visuals which would ordinarily occour in the receiver client. The receiver is not involved in the process until the drop stage begins. But it has to supply information about drop sites so the initiator can handle the drag-under visuals accordingly. This mode minimises network trafic, but the drop site can't determine whether it wants to accept the drop data until the drop actually occurs. In addition, the server is grabbed during the drag operation.

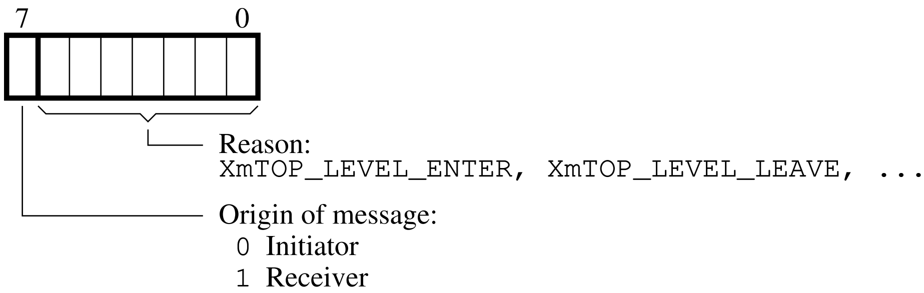
### 6.2.2 Protocol Messages

Unfortunately, X client messages (of type `XClientMessageEvent`) are limited in size: you can only transfer up to 20 bytes within them. For this reason, M*TIF applications use a "drag window" with special properties attached to it to transfer any additional information which uses up to much space. Some of the protocol messages therefore just refer to a data structure stored with the drag window.

The various X client messages used for the drag or drop protocol share a common header and some common settings.

- The `message_type` of such an X client message event is always set to the atom with the name `_MOTIF_DRAG_AND_DROP_MESSAGE`.
- The `format` of the message event is `8`, so no byte swapping is performed by the X server. The clients participating in the drag and drop mechanism must do the byte swapping themselves. The reason for this is that the available 20 bytes can be exploited best this way.

- The `window` identifier of the `XClientMessageEvent` contains the identifier of the window receiving this message.
- The first user byte `data.b[0]` of the message event indicates why this message was sent. It is a bitfield with the high bit (bit 7) indicating whether the originator or the receiver generated the message, and the remaining bits 6–0 denoting the reason. The reason can be anyone of the message types listed in table 6.1. These message types are discussed in detail in the sections below.




| Identifier | Value |
|---|---|
| XmTOP_LEVEL_ENTER | 0x00 |
| XmTOP_LEVEL_LEAVE | 0x01 |
| XmDRAG_MOTION | 0x02 |
| XmDROP_SITE_ENTER | 0x03 |
| XmDROP_SITE_LEAVE | 0x04 |
| XmDROP_START | 0x05 |
| XmDROP_FINISH | 0x06 |
| XmDRAG_DROP_FINISH | 0x07 |
| XmOPERATION_CHANGED | 0x08 |

**Table 6.1:** *Message types used for the Drag & Drop X client messages.*

- `data.b[1]` indicates the byte order used for the encoding of the following data. Like in the X Protocol it must be set to either the ASCII uppercase letter ‘`B`’ when the *most*-significant byte is transmitted first, or to the ASCII lowercase letter ‘`l`’ when the *least*-significant byte is transmitted first.
- The data bytes `data.b[2]` through `data.b[19]` contain the remaining bytes of the drag and drop message.

### 6.2.3 Drag & Drop Flags

Many of the drag and drop messages contain a flag bitfield, which is called the “DnD Flags” throughout this documentation. The “DnD Flags” consist of four distinct bitfields – each of it is four bits wide and is labeled ① through ④ in figure 6.1.

Not every message makes use of every of the four bitfields. Some drag and drop messages don’t use the “DnD Flags” at all, although they contain a spare field with the same size and in the same position as the "DnD Flags". Below, the descriptions of the various drag and drop messages will refer to this bitfields ① through ④. Thus, you can easily tell, which bitfields are used for a particular message.

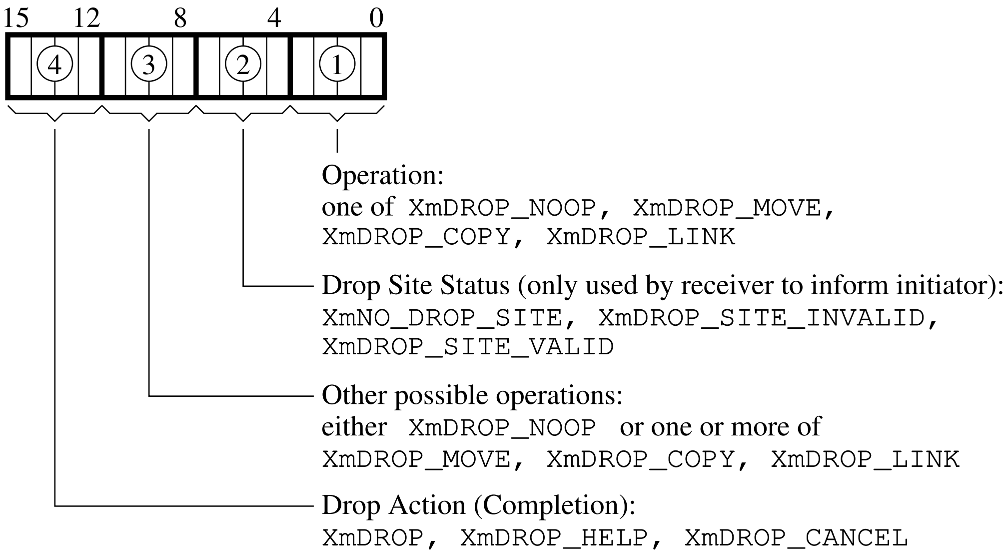



**Figure 6.1:** *The "DnD Flags" signal various status conditions during a drag and drop operation.*

Their purpose is as follows:

① Operation: this bitfield contains the *recommended* type of drag and drop operation, if it is used in a message sent by the initiator. If the message was sent instead by the receiver, then this bitfield contains the *selected* type of drag and drop operation. Thus, the receiver is free to "dictate" the kind of drag and drop operation. See table 6.2 for possible values.

② Drop Site Status: this bitfield is only used by the receiver to inform the initiator whether the pointer is currently hoovering over a valid/invalid drop site or no drop site at all. See table 6.4 for possible values.

③ Other possible operations: this is a bitset (binary OR) of all the operations that can be carried out on the current drop site. If the pointer isn't currently over a valid drop site, then this bitset has all its bits set to zero (`XmDROP_NOOP`). See table 6.2 for the values of the various flags.

④ Drop Action: this bitfield is used only when starting the *drop* operation. In this particular case it is used by the receiver to indicate the drop action which the receiver is going to carry out. See table 6.3 for valid completion status codes.

| Identifier | Value |
|---|---|
| `XmDROP_NOOP` | `0x00` |
| `XmDROP_MOVE` | `0x01` |
| `XmDROP_COPY` | `0x02` |
| `XmDROP_LINK` | `0x04` |

**Table 6.2:** *Operation codes.*

| Identifier | Value |
|---|---|
| `XmDROP` | `0x00` |
| `XmDROP_HELP` | `0x01` |
| `XmDROP_CANCEL` | `0x02` |
| `XmDROP_INTERRUPT` | `0x03` |

**Table 6.3:** *Completion status codes.*

| Identifier | Value |
|---|---|
| XmNO_DROP_SITE | 0x01 |
| XmDROP_SITE_INVALID | 0x02 |
| XmDROP_SITE_VALID | 0x03 |
| XmINVALID_DROP_SITE (DEPRECATED SYMBOL) | 0x02 |
| XmVALID_DROP_SITE (DEPRECATED SYMBOL) | 0x03 |

**Table 6.4:** *Drop site status codes.*

### 6.2.4 The Targets Table

The M*TIF toolkit uses a special persistent, input-only, and override-redirected window to store some data needed for the whole drag and drop infrastructure. This special window is a child of the display's default root window and is called the "M*TIF Drag Window". In order that clients can find this window at all, they should watch out for the property `_MOTIF_DRAG_WINDOW` (of type `WINDOW`, with a size of 32) on their display's default root window. If there's no such property, or the window ID stored in the property is either `0` or invalid, then an client should create the "M*TIF Drag Window" itself. The "M*TIF Drag Window" should have a close-down mode of `RetainPermanent` (use `XSetCloseDownMode()` for this), so other applications don't have to create it themselves.

The "M*TIF Drag Window" currently seems to posses three properties named `_MOTIF_DRAG_TARGETS` (of type `_MOTIF_DRAG_TARGETS`, size is 8), `_MOTIF_DRAG_ATOMS` (of type – guess which – `_MOTIF_DRAG_ATOMS`), and finally `_MOTIF_DRAG_ATOM_PAIRS` (of course of the type `_MOTIF_DRAG_ATOM_PAIRS`).

Of primary interest to us is the `_MOTIF_DRAG_TARGETS` property: it is a list of target lists, which is shared among all clients, and is commonly called the "targets table". Every target list within the targets table is a list of target (data types) an initiator can supply to a receiver on request. Remember, that the X client messages only provide precious little space for the client's data. Thus, instead of passing such lists around, you only need to pass a single `CARD16` value that acts as an index (0-based) into the list of target lists. By specifying such an index, a receiver knows which kinds of data it can request from the initiator during the drop phase.

Whenever a client needs to add his target list(s) to the targets table, it must follow some guidelines, otherwise the targets table could become messed up. Every target list must be sorted into ascending order (according to the atom ID's) to avoid permutations of otherwise compatible target lists. Thus, if a client supports the target types "B,A,C" and another client supports the target types "C,B,A", then they must both use the sorted target types list "A,B,C" instead. Note however that the targets `TARGETS` and `MULTIPLE` – which are mandatory according to the ICCCM – are never listed. The structure of the targets table is fairly straightforward and is shown in table 6.5.

In order to add its target list(s) to the targets table stored in the `_MOTIF_DRAG_TARGETS` property, a client must grab the X server, so the operation is atomically. Then it has to search the targets table for a match. Otherwise, the client can add the particular (sorted) target list to the table anywhere

| Offset | Size | Description |
|---|---|---|
| +0x00 | BYTE | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| +0x01 | BYTE | **Protocol Version**: currently 0. |
| +0x02 | CARD16 | **Number of Target Lists**: this should be really self-explanatory. |
| +0x04 | CARD32 | **Data Size**: total size of the data stored in the property:<br>8 bytes for the header +<br>Number of Target Lists * 2 +<br>Total Number of Targets * 4 |
| +0x08 | CARD16 | **Number of Targets in List** |
| +0x0A | CARD32 | **List of Targets** |
| +0x?? | CARD16 | **Number of Targets in List** |
| +0x?? | CARD32 | **List of Targets** |
| | | *...and so on...* |

**Table 6.5:** *Structure of the targets table in the property _MOTIF_DRAG_TARGETS.*

(the target table itself is not sorted – sigh). After it has updated the _MOTIF_DRAG_TARGETS property, the client can remove the grab.

### 6.2.5 Advertising a Receiver

A receiver advertises itself by placing a property with the name _MOTIF_DRAG_RECEIVER_INFO (of type _MOTIF_DRAG_RECEIVER_INFO, size is 8) on its top level window. Depending on the protocol styles the receiver can handle (dynamic mode and/or preregister mode), this property contains more or less data. But at least the property must be 16 bytes long.

| Offset | Size | Description |
|---|---|---|
| +0x00 | BYTE | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| +0x01 | BYTE | **Protocol Version of Receiver**: currently 0. |
| +0x02 | BYTE | **Protocol Style**: one of the protocol styles listed in table 6.7 |
| +0x03 | BYTE | **Padding**. |
| +0x04 | CARD32 | **Proxy Window**: – under construction – |
| +0x08 | CARD16 | **Number of Drop Sites**: number of drop site blocks, which are immediately following this header. |
| +0x0A | CARD16 | **Padding**. |
| +0x0C | CARD32 | **Total Size**: – under construction – |

**Table 6.6:** *The structure of the _MOTIF_DRAG_RECEIVER_INFO property describes a drag and drop receiver.*

The second half of the header shown in table 6.6 is only used in the preregister mode. For the dynamic mode and when the drop site is a drop-only site, then the second half within the header is not needed. Because the preregister mode is really tricky, we're discussing it later in more detail.

| Identifier | Value |
|---|---|
| XmDRAG_NONE | 0x00 |
| XmDRAG_DROP_ONLY | 0x01 |
| XmDRAG_PREFER_PREREGISTER | 0x02 |
| XmDRAG_PREREGISTER | 0x03 |
| XmDRAG_PREFER_DYNAMIC | 0x04 |
| XmDRAG_DYNAMIC | 0x05 |
| XmDRAG_PREFER_RECEIVER | 0x06 |

**Table 6.7:** *Drag protocol styles.*

When a receiver signals that it can handle the dynamic mode, then it will accept and answer the drag messages sent by the initiator as specified in the next section. If the protocol style of the receiver is drop-only, then the initiator should not send any drag messages. The only message the initiator is allowed to send then is the `XmDROP_START` message. In this case the visual effects during the drag and drop gesture should indicate to the user that the whole top level window of the receiver works as a single drop site accepting all possible targets and operations.

A receiver with a protocol style of `XmDRAG_NONE` wants not to be disturbed by drag and drop operations at all. The initiator should provide appropriate visual feedback whenever the drag icon is over such a receiver.

### 6.2.6 Starting a Drag or Drop

When an initiator starts a drag – or even only a drop –, then it first creates a new property on the source window. The name of this property can be arbitrary, but it must be of type `_MOTIF_DRAG_INITIATOR_INFO` and have a data size of 8. This property then should contain information about which targets the initiator is able to serve and what selection atom to use for the data transfer. Table 6.8 shows the structure of this property.

| Offset | Size | Description |
|---|---|---|
| +0x00 | BYTE | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| +0x01 | BYTE | **Protocol Version of Initiator**: currently 0. |
| +0x02 | CARD16 | **Targets Index**: the index of a targets list within the targets table. The first list within the targets list has an index of 0. This index advertises which targets the initiator is willing to handle. |
| +0x04 | CARD32 | **Selection Atom**: atom ID to be used as the selection atom for the data transfer when the drop actually takes place. |

**Table 6.8:** *The structure of the `_MOTIF_DRAG_INITIATOR_INFO` property describes the initiator.*

Figure 6.2 finally shows the four properties which are representing an important part of the information infrastructure for the drag and drop mechanism.

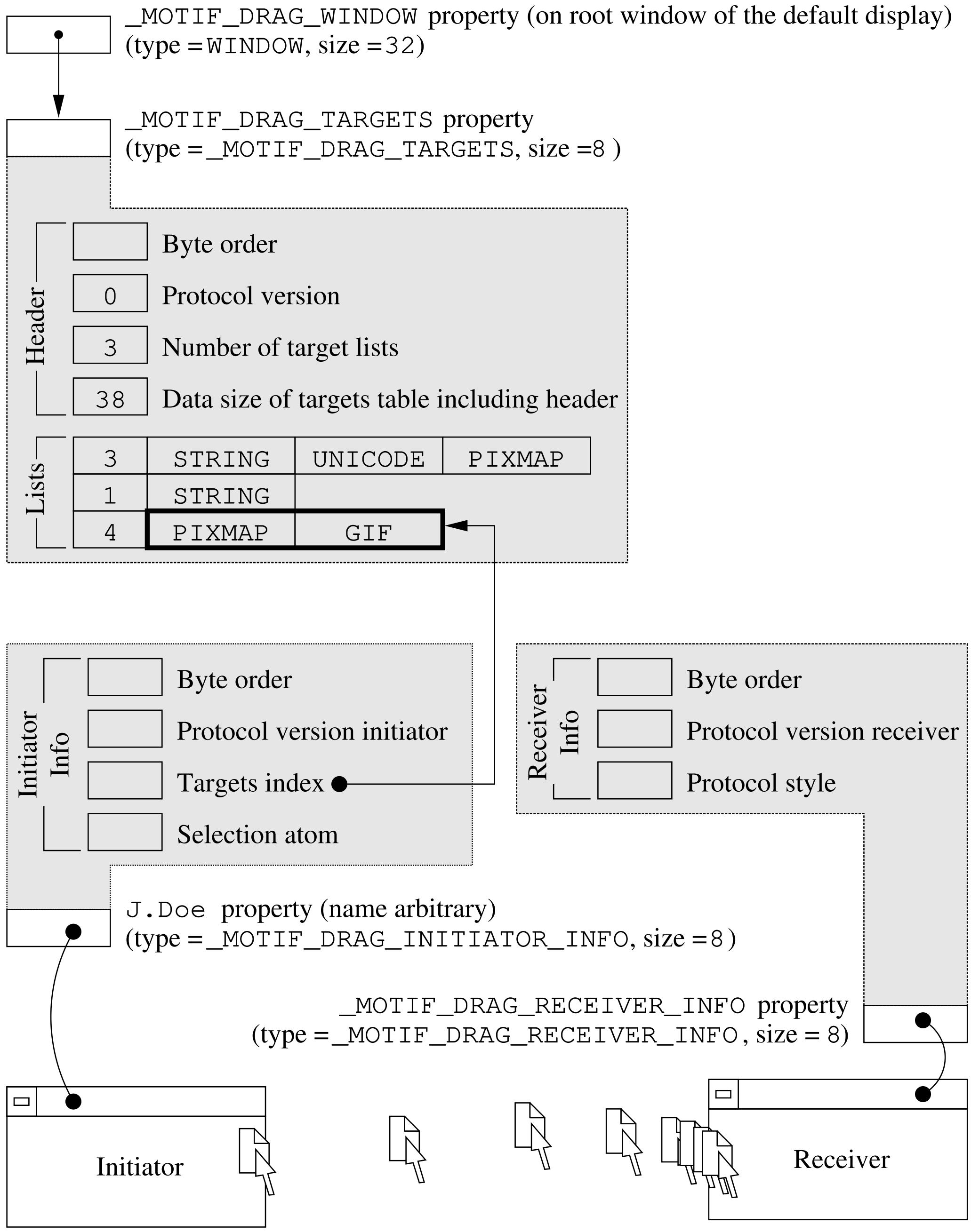



**Figure 6.2:** *Properties involved in the Drag & Drop game.*

## 6.3 The Drag Protocol

During a drag operation the client is free to skip the drag protocol. Reasons for this might be either complexity or known latencies. But in turn the client looses the ability to provide accurate (dynamic) feedback during the drag.

### 6.3.1 Entering/Leaving Top Level Windows

When the pointer enters a new top level window, the initiator notifies the receiver with a `XmTOP_LEVEL_ENTER` message.

| Message User Data | Size | Description |
|---|---|---|
| `data.b[0]` | `BYTE` | **Reason**: `XmTOP_LEVEL_ENTER` (`0x00`) |
| `data.b[1]` | `BYTE` | **Byte Order**: either ’`B`’ (MSB first) or ’`l`’ (LSB first). |
| `data.b[2..3]` | `CARD16` | (**DnD Flags**: unused) |
| `data.b[4..7]` | `CARD32` | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |
| `data.b[8..11]` | `CARD32` | **Source Window**: source window ID of the initiator. |
| `data.b[12..15]` | `CARD32` | **Drag Initiator Info Atom**: atom ID of a property the initiator set up when it started the drag or drop operation. |

**Table 6.9:** *The `XmTOP_LEVEL_ENTER` message send by the initiator.*

The atom ID (in table 6.9 it is called the “Drag Initiator Info Atom”) sent in the `XmTOP_LEVEL_ENTER` drag message is a selection atom. It must be unique for the duration of the Drag & Drop transaction. In addition, the initiator must own the selection and must be ready to convert data from the early beginning of the drag operation, since the receiver can ask for a conversion dynamically during the drag to validate the operation.

When the pointer leaves a top level window, the initiator notifies the receiver with a `XmTOP_LEVEL_LEAVE` message.

| Message User Data | Size | Description |
|---|---|---|
| `data.b[0]` | `BYTE` | **Reason**: `XmTOP_LEVEL_LEAVE` (`0x01`) |
| `data.b[1]` | `BYTE` | **Byte Order**: either ’`B`’ (MSB first) or ’`l`’ (LSB first). |
| `data.b[2..3]` | `CARD16` | (**DnD Flags**: unused) |
| `data.b[4..7]` | `CARD32` | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |

**Table 6.10:** *The `XmTOP_LEVEL_LEAVE` message send by the initiator.*

A receiver never replies (echoes) the `XmTOP_LEVEL_ENTER` and `XmTOP_LEVEL_LEAVE` messages.

### 6.3.2 Pointer Motion

When the pointer moves, the initiator sends `XmDRAG_MOTION` messages to the receiver (which is the top level window the pointer is currently in and which is willing to accept drag messages).

| Message User Data | Size | Description |
|---|---|---|
| `data.b[0]` | `BYTE` | **Reason**: `XmDRAG_MOTION (0x02)` |
| `data.b[1]` | `BYTE` | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| `data.b[2..3]` | `CARD16` | **DnD Flags**: ① + ③ |
| `data.b[4..7]` | `CARD32` | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |
| `data.b[8..9]` | `CARD16` | **Root X**: x position of the drag-over icon relative to the root window. |
| `data.b[10..11]` | `CARD16` | **Root Y**: y position relative to the root window. |

**Table 6.11:** *The `XmDRAG_MOTION` message send by the initiator.*

Whenever the initiator sends a `XmDRAG_MOTION` message, the receiver responds with one out of three different messages, depending on whether the pointer entered or left a valid drop site (`XmDROP_SITE_ENTER`, `XmDROP_SITE_LEAVE`), or just moved around (`XmDRAG_MOTION`).

| Message User Data | Size | Description |
|---|---|---|
| `data.b[0]` | `BYTE` | **Reason**: `XmDROP_SITE_ENTER (0x83)` |
| `data.b[1]` | `BYTE` | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| `data.b[2..3]` | `CARD16` | **DnD Flags**: ① + ② + ③ |
| `data.b[4..7]` | `CARD32` | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |
| `data.b[8..9]` | `CARD16` | **Root X**: better x position (hint for the initiator) of the drag-over icon relative to the root window. |
| `data.b[10..11]` | `CARD16` | **Root Y**: better y position relative to the root window. |

**Table 6.12:** *The `XmDROP_SITE_ENTER` message replied by the receiver.*

| Message User Data | Size | Description |
|---|---|---|
| `data.b[0]` | `BYTE` | **Reason**: `XmDROP_SITE_LEAVE (0x84)` |
| `data.b[1]` | `BYTE` | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| `data.b[2..3]` | `CARD16` | (**DnD Flags**: unused) |
| `data.b[4..7]` | `CARD32` | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |

**Table 6.13:** *The `XmDROP_SITE_LEAVE` message replied by the receiver.*

| Message User Data | Size | Description |
|---|---|---|
| data.b[0] | BYTE | **Reason**: XmDRAG_MOTION (0x82) |
| data.b[1] | BYTE | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| data.b[2..3] | CARD16 | **DnD Flags**: ① + ② + ③ |
| data.b[4..7] | CARD32 | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |
| data.b[8..9] | CARD16 | **Root X**: better x position (hint for the initiator) of the drag-over icon relative to the root window. |
| data.b[10..11] | CARD16 | **Root Y**: better y position relative to the root window. |

**Table 6.14:** *The XmDRAG_MOTION message echoed by the receiver.*

### 6.3.3 Changing the Operation

The user is free to change the drag operation (copy, move, link) at any time during the drag gesture – for example, if she/he presses or releases modifier keys. The initiator then sends an XmOPERATION_CHANGED message.

| Message User Data | Size | Description |
|---|---|---|
| data.b[0] | BYTE | **Reason**: XmOPERATION_CHANGED (0x08) |
| data.b[1] | BYTE | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| data.b[2..3] | CARD16 | **DnD Flags**: ① + ③ |
| data.b[4..7] | CARD32 | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |

**Table 6.15:** *The XmOPERATION_CHANGED message send by the initiator.*

The receiver then echoes the XmOPERATION_CHANGED message.

| Message User Data | Size | Description |
|---|---|---|
| data.b[0] | BYTE | **Reason**: XmOPERATION_CHANGED (0x88) |
| data.b[1] | BYTE | **Byte Order**: either 'B' (MSB first) or 'l' (LSB first). |
| data.b[2..3] | CARD16 | **DnD Flags**: ① + ② + ③ |
| data.b[4..7] | CARD32 | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |

**Table 6.16:** *The XmOPERATION_CHANGED message echoed by the receiver.*

## 6.4 The Drop Protocol

The drop protocol can be used independently of the drag protocol, for example when a drop site is in the `XmDRAG_DROP_ONLY` mode. Fortunately, the whole drop protocol is really lean – it just consists of the single message `XmDROP_START` sent by the initiator, which must be echoed by the receiver.

| Message User Data | Size | Description |
|---|---|---|
| `data.b[0]` | `BYTE` | **Reason**: `XmDROP_START` (`0x05`) |
| `data.b[1]` | `BYTE` | **Byte Order**: either `'B'` (MSB first) or `'l'` (LSB first). |
| `data.b[2..3]` | `CARD16` | **DnD Flags**: ① + ③ |
| `data.b[4..7]` | `CARD32` | **Timestamp**: set to the timestamp of the corresponding X event triggering this message. |
| `data.b[8..9]` | `CARD16` | **Root X**: x position relative to the root window. |
| `data.b[10..11]` | `CARD16` | **Root Y**: y position relative to the root window. |
| `data.b[12..15]` | `CARD32` | **Drag Initiator Info Atom**: atom ID of a property the initiator set up when it started the drag (or drop, if we're in `XmDRAG_DROP_ONLY` mode). |
| `data.b[16..19]` | `CARD32` | **Source Window**: source window ID of the initiator. |

**Table 6.17:** *The `XmDROP_START` message send by the initiator.*

The receiver then echoes this `XmDROP_START` message, indicating whether it is willing to accept the drop and what operation (move, copy or link) it want to carry out.

| Message User Data | Size | Description |
|---|---|---|
| `data.b[0]` | `BYTE` | **Reason**: `XmDROP_START` (`0x85`) |
| `data.b[1]` | `BYTE` | **Byte Order**: either `'B'` (MSB first) or `'l'` (LSB first). |
| `data.b[2..3]` | `CARD16` | **DnD Flags**: ① + ② + ③ + ④ |
| `data.b[4..5]` | `CARD16` | **Better X**: better x position (hint for the initiator) of the drag-over icon relative to the root window. |
| `data.b[6..7]` | `CARD16` | **Better Y**: better y position (hint for the initiator) of the drag-over icon relative to the root window. |

**Table 6.18:** *The `XmDROP_START` message echoed by the receiver.*

If the receiver did not cancel the drop (which it must indidicate within the echoed message), then it can proceed to transfer the drop data using the X selection transfer mechanism. The receiver can request as many transfers as it wants, using the selected targets. For each conversion request, the initiator replies using the ICCCM selection.

The resources allocated during the drag operation should not be released until the drop is finished. The receiver indicates this by requesting a conversion for the targets `XmTRANSFER_SUCCESS` or `XmTRANSFER_FAILURE`. When the initiator receives such a conversion request, then it must reply with an empty value.The receiver as well as the initiator can then release all resources allocated for the drop operation. In addition, it's the right time to show the melting or failure/snapback visual effect.

## 6.5 The Preregister Mode

The full protocol described in this chapter so far is only used when both the initiator and receiver have agreed to use the dynamic mode. When using the preregister mode, the initiator grabs the server when the drag starts, and whenever the pointer enters a new top level window, it reads all the drop site information it needs for doing its tracking, visual feedback, etc., from a preregistered database attached to the `_MOTIF_DRAG_RECEIVER_INFO` property of each participating client top level window. Obviously, no drag X client messages are sent, since no one is listening to them (remember, that the X server is grabbed). The server gets ungrabbed when the user drops the object, at which point the documented drop protocol comes in effect (together with the convention for the transfer success or failure).

For the preregister mode, the `_MOTIF_DRAG_RECEIVER_INFO` property has also valid information stored in the second half, as descibed in table 6.6. In addition, the header (which has a size of 16 bytes) is followed by "drop site blocks" that describe the drop sites located within the top level window of a receiver.

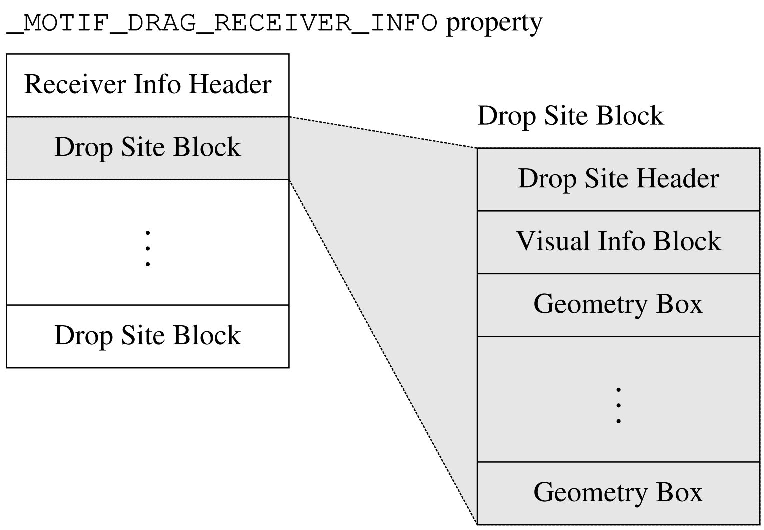



**Figure 6.3:** *Overall structure of the* `_MOTIF_DRAG_RECEIVER_INFO` *property for the preregister mode.*

For each drop site there is a corresponding drop site block in the `_MOTIF_DRAG_RECEIVER_INFO` property. Each drop site block starts with a 8 bytes long "drop site header", and is followed by a "visual info block" and a series of geometry boxes giving geometry information about a drop site. This overall structure is shown in figure 6.3.

| Offset | Size | Description |
|---|---|---|
| +0x00 | CARD16 | **Drop Site Flags**: bitfield containing various flags which describe the possible operations, the drop type, the animation style, as well as some other things. |
| +0x02 | CARD16 | **Targets Index**: the index of a targets list within the targets table. The first list within the targets table has an index of 0. This index advertises which targets the initiator is willing to handle. This value is used in the same way as the Targets Index of the _MOTIF_DRAG_INITIATOR_INFO property. |
| +0x04 | CARD32 | **Number of Geometry Boxes**: the number of geometry boxes following the visual info block for this drop site block. |

**Table 6.19:** *The structure of a drop site block header*

– under construction –

# 7

# When the Keyboard Goes Wild

Harald Albrecht

## 7.1 Introduction

To some extent, the way X treats keyboard input is more complicated than handling the pointing device events (or many other events). The information about a key in the event structure isn't suitable for immediate use, instead it has to go through one of several conversion stages before it becomes useful to the application. The main reason is that X is designed to support all kinds of keyboards. However, the drawback is increased complexity.

Keyboard input appears in three "flavours": as keycodes, keysyms or keystrings. Naturally, M*TIF adds a fourth one: the CSF keysyms. They are a special set of keysyms. Figure 7.1 shows how keycodes, keysyms and keystrings relate.

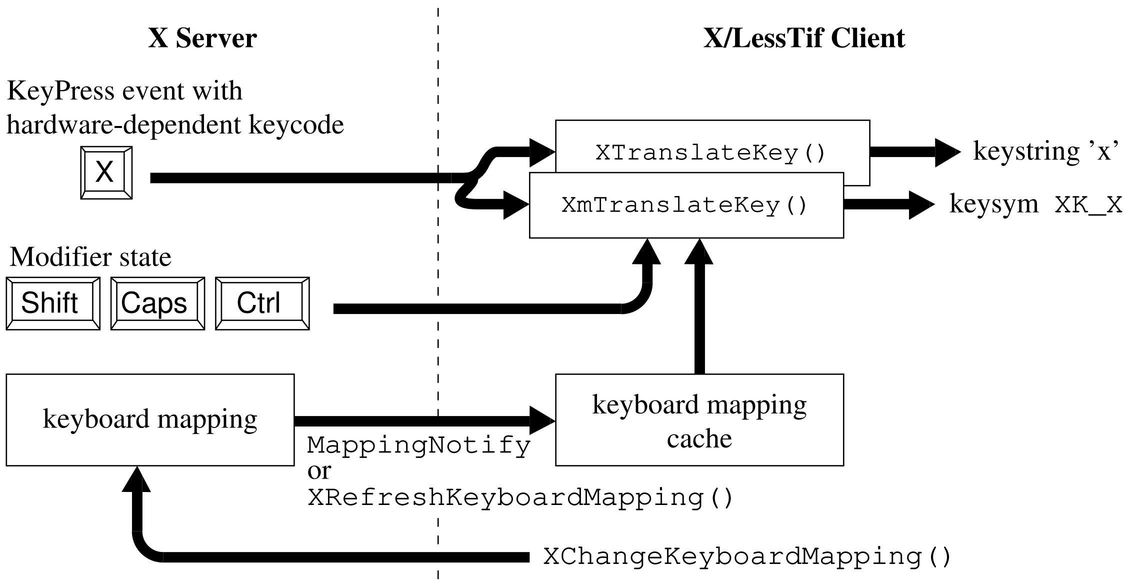



**Figure 7.1:** *Keyboard event processing.*

A keycode is a hardware-dependent coding of the key being pressed or released. Thus, keycodes aren't really usefull to LESSTIF application writers. With the help of the keyboard mapping, every Xlib and/or Xt intrinsics client converts the hardware-dependent keycodes into hardware-independent keysyms. These keysyms are integers representing the symbol engraved on a key ("a", "A", "+" as well as "Shift", "PageDn" and other ones). The keyboard mapping is read from the server and cached to speed up look-ups and prevent unnecessary round-trips to the server. The translation from keycodes to keysyms can be done with `XtTranslateKeycode()`. This calls the currently registered key translator procedure, which is `XmTranslateKey()` by default. The translation manager calls the key translator procedure, too.

Keysyms that represent printable characters can be further translated into keystrings by calling `XTranslateKey()`. This mapping between keysyms and keystrings is not stored in the X server, but rather hardwired into the Xlib.

## 7.2 The Virtual Bindings

Unfortunately the concept of keysyms leaves too much room for vendor-dependent interpretations on how to bind keycodes to keysyms. A constant source for confusion and frustration is the key ← at the top right of the alphanumeric key area. Some vendors bind this key to the keysym `XK_BackSpace`, some others to the keysym `XK_Delete`.

The simplest way would be to change the keyboard mapping of the server. But this would affect all clients connected to that server – especially the non-LESSTIF ones. In order to reassert some (basic) consistency, the CSF introduced the "CSF keysyms". The CSF keysyms form a special set of keysyms. Depending on the X server's vendor, certain keycodes are translated into CSF keysyms. The mappings between keysyms and CSF keysyms are also known as the "virtual bindings" (see figure 7.2). The conversion between keypresses and keysyms takes place in `XmTranslateKey()`.

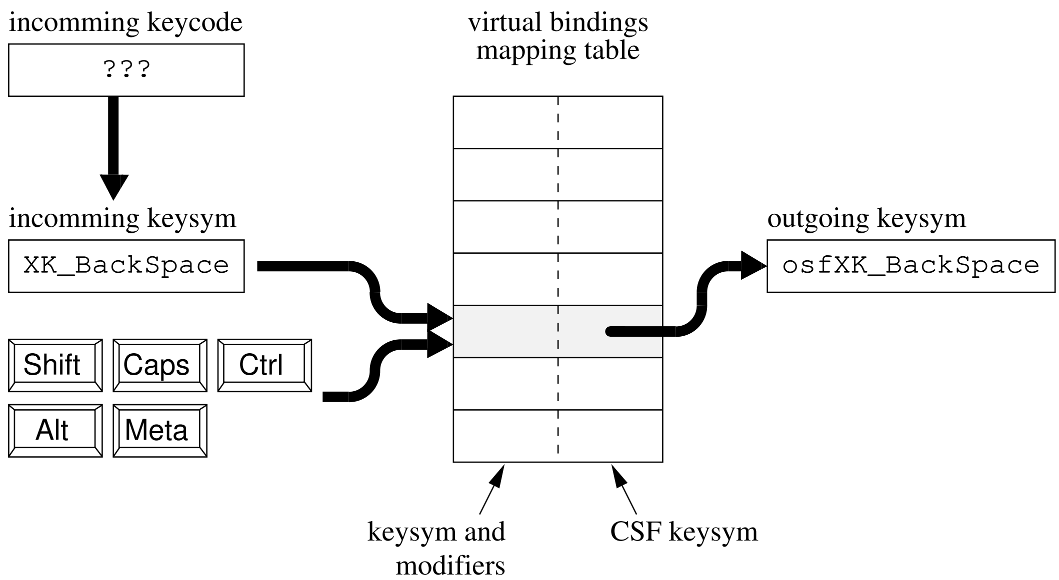



**Figure 7.2:** *The "virtual bindings" take care of some basic consistency between different keyboards.*

When converting ordinary innocent keysyms into CSF keysyms the modifiers must be taken into consideration. Unfortunately, we have to cope with different kinds of modifiers when looking at the virtual binding mechanism or the translation manager. An example shall enlighten the problem arising from this. Suppose your new WYSIWYWAA widget (WHAT YOU SEE ISN'T WHAT YOU WANTED AFTER ALL) also features an "undo" operation. The undo is activated by pressing the keys ALT and ← together. Somewhere in your widget's translation table you'll have to write:

```
Alt<Key>osfDelete:  undo()
Meta<Key>osfDelete: undo()
```

As LESSTIF – or its "alter ego" M*TIF – converts some of the standard keysyms into virtual keysyms, you can't use the standard keysym name `Delete` but must use `osfDelete`. The translation manager will never see the original keysym but only the virtual keysym.

Although the "`Alt`" modifier preceeding the keypress translation "`<Key>`" looks suspiciously like any ordinary modifier (e.g., `Shift`), it isn't one! X provides for eight modifiers alltogether, but only `Shift`, `Lock`, and `Control` are predefined. The remaining five modifiers `Mod1` up to `Mod5` can be freely mapped to any key. Smart as the Xt intrinsics are, they convert the translation `Alt<Key>osfDelete` into a translation using the Alt *key* instead of a (ficious) "Alt" *modifier*. Unfortunately, many programmers aren't aware of this.

Thus, when looking at translations, it is very important to distinct between the two sets of *real* modifiers and *fake* modifiers: The real modifiers are: `Shift`, `Lock`, `Ctrl`, `Mod1` up to `Mod5`, and the mouse buttons `Button1` up to `Button5`, whereas fake modifiers are: `Alt`, `Meta`, `Super`, and `Hyper`.

During the process of converting keysyms into CSF keysyms (in `XmTranslateKey()`) no translation mechanism is present. All we have is the current state of the modifiers recorded in the KeyPress event. But only `Shift`, `Lock`, and `Control` are predefined. So what to do with virtual bindings which are supposed to translate `Alt`'ed keysyms into CSF keysyms?

The CSF decided to ignore the problem as good as possible when developing M*TIF, and depends on the user mapping the Alt key on the keyboard to the `Mod1` modifier. You can test this by changing your mapping such, that Alt maps to `Mod2`. The respective virtual bindings won't work any longer.

LESSTIF is much smarter than M*TIF (What?! Impossible!). Unfortunately, the `XmTranslateKey()` converter can't maintain the current keyboard state of the Alt and Meta keys as it gets called from the translation manager many times even for every *single* keystroke. Thus, during startup LESSTIF tries to find out to which modifier the Alt key has been bound to. LESSTIF does this basically by scanning the modifier mapping (as returned by `XGetModifierMapping()`) for the modifier bound to the keysyms `XK_Alt_L` or `XK_Alt_R`. If none can be found LESSTIF falls back to use `Mod1`. The whole procedure is so easy to implement, I can't understand why the CSF didn't get the trick in the past. If you're interested how LESSTIF does it, take a look at the file `$(LESSTIF_ROOT)/libXm/VirtKeys.c`. The source is commented (really!).

## 7.3 Managing the Modifier Mappings

There are three functions available for messing about with the current mapping of the `Alt` modifier (remember that these ones are LESSTIF-specific!):

```
XmModifierMaskSetReference _XmGetModifierMappingsForDisplay(Display *dpy);
void _XmInvalidateModifierMappingsForDisplay(Display *dpy);
void _XmRefreshVirtKeys(Widget w);
```

You'll normally use only `_XmGetModifierMappingsForDisplay()`. This function reports the current mapping as a pointer to a `XmModifiersMaskSet` (and surely gets an olympic medal for its name's length). This set is simply an array that holds the modifier masks for the Alt, Meta, Super, and Hyper modifier keys. If LESSTIF can't find a binding for the Alt key it will fall back to the `Mod1` modifier mask as the `Alt` modifier mask.

To get the modifier mask of the `Alt` modifier, just use the index `ALTModifier` into the array:

```
#ifdef LESSTIF_VERSION
#include <Xm/VirtKeysP.h>
    XmModifierMaskSetReference ModifierMasks;
#endif
    Modifiers                  Alt, someModifierFlags;

#ifdef LESSTIF_VERSION
    ModifierMasks = _XmGetModifierMappingsForDisplay(dpy);
    Alt           = ModifierMasks[ALTModifier];
#else
    Alt           = Mod1Mask;
#endif

    someModifierFlags = ... ;
    if ( someModifierFlags & Alt ) {
        .... ;
    }
```

The result of `_XmGetModifierMappingsForDisplay()` is cached so all but the first request won't result in a round-trip to the X server. The modifier mask set belongs to the cache, so be sure to never free it.

When the user changes the modifier mapping during the lifetime of a LESSTIF based application, LESSTIF receives a MappingNotify event and updates its modifier cache as well as the virtual bindings by calling `_XmRefreshVirtKeys()`. If for any reason you must invalidate the modifier mapping cache, you can call `_XmInvalidateModifierMappingsForDisplay()`. Any pointer to the modifier mapping array for the respective display then gets invalid!

## 7.4 Managing the Virtual Bindings

The virtual binding mechanism in `VirtKeys.c` provides four additional ("undocumented") functions to mess with:

```
void _XmVirtualToActualKeysym(Display *Dsp, KeySym VirtualKeysym,
                              KeySym *RealKeysymReturn,
                              Modifiers *ModifierReturn);
void _XmVirtKeysInitialize(Widget w);
Boolean _XmVirtKeysLoadFileBindings(String filename, String *binding);
int _XmVirtKeysLoadFallbackBindings(Display *Dsp, String *Bindings);
```

These four functions are available with M*TIF as well as with LESSTIF. With `_XmVirtualTo-ActualKeysym()` you can check how a virtual keysym would look like in real life. You also get back from the function the necessary modifiers which must be active in order to convert the real keysym into a CSF keysym.

The contents of a file can be loaded into memory by means of the _XmVirtKeysLoadFileBindings() function. The memory needed to hold the contents is allocated by the function and must be freed when it's not needed any more. If the function fails for any reason (file not found, not enough memory available) _XmVirtKeysLoadFileBindings() returns False.

If for any reason you need to set up the _MOTIF_DEFAULT_BINDINGS property of the root window of a given display, you can use _XmVirtKeysLoadFallbackBindings() for this task. If applicable, the function will load a vendor-specific set of virtual bindings. Otherwise it will fall back to a generic set of virtual bindings. _XmVirtKeysLoadFallbackBindings() returns in the parameter Bindings the current set of virtual bindings. You are responsible for freeing the string with XtFree() when you don't need it any longer. The most interesting use of this function is within the xmbind client. If no binding file is specified and there is no .motifbind file available, then xmbind can install the default fallback bindings in the _MOTIF_DEFAULT_BINDINGS property. More on this in the next section.

You will hardly need to call _XmVirtKeysInitialize(), as this sets up the virtual bindings on a XmDisplay widget. It gets automatically called during the initialising phase of this kind of widget. This function is solely for use within the LESSTIF modules VirtKeys.c and Display.c.

## 7.5 The xmbind Client

The current virtual bindings are stored in one of two possible properties on the root window of *screen* #0. There can be only one set of active virtual bindings at the same time on a given *display* as there exists only *one* keyboard per display. Therefore the current bindings are always stored in a property of the root window of screen #0. Please note that they are *not* attached to the root window of the default screen, as the default screen can be any screen of a given display and may even change from application to application (see figure 7.3). The storage for the current virtual bindings is provided by one of the following properties (both of type XA_STRING):

- The property _MOTIF_DEFAULT_BINDINGS (if existent) contains the default virtual bindings for the display.
- The property _MOTIF_BINDINGS contains virtual bindings loaded either from the user's $(HOME)/.motifbind file or from xmbind.alias (available in several good places).

If none of the two properties exist, LESSTIF's startup-code first tries to find user-specific bindings and if it succeeds, it sticks them to the _MOTIF_BINDINGS property. Otherwise the starup-code figures out the default virtual bindings (according to the display) and loads them into the _MOTIF_DEFAULT_BINDINGS property. In every case, after starting a LESSTIF application the root window of screen #0 contains a property specifying the current virtual bindings.

The xmbind client can be used to change or setup the properties related to the virtual bindings. This client is remarkable simple, as most of the functionality needed is already laid down in the

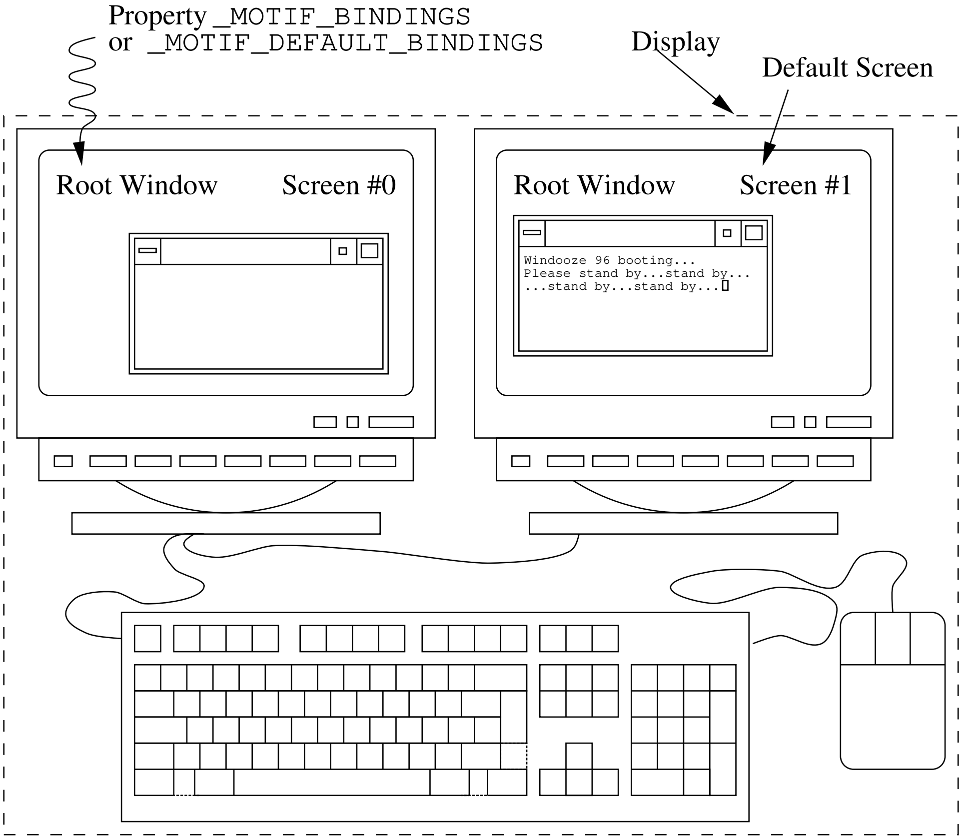



**Figure 7.3:** *Displays, Screens and the Virtual Bindings.*

LESSTIF library (mostly in the form of "undocumented" functions). Following is the pseudo-code of `xmbind` (the source resides in `$(LESSTIF_ROOT)/clients/xmbind/xmbind.c`):

```
if (user specified a file on the command line) {
    delete the _MOTIF_DEFAULT_BINDINGS property
    load the file into the _MOTIF_BINDINGS property
} else {
    if (there is a .motifbind file) {
        delete the _MOTIF_DEFAULT_BINDINGS property
        load the file into the _MOTIF_BINDINGS property
    } else {
        delete the _MOTIF_BINDINGS property
        load fallback bindings into the  _MOTIF_DEFAULT_BINDINGS property
    }
}
flush the connection to the display and terminate
```

# 8

# Inside XmStrings

Chris Toshok
Harald Albrecht

## 8.1 Introduction

This chapter gives a cursory explanation of the way XmStrings are encoded in M*TIF 1.2 (and LESSTIF). This information is still being discovered, so explanations of where it is wrong are welcome.

Sometime last year on the LESSTIF mailing list, a well known SGI persona, Doug Rand, sent an email that described the things that were changed from 1.2 to 2.0. One of the things he mentioned was that the encoding rules for the external representation of XmStrings can no longer be considered to be in ASN.1 format (ASN means "Abstract Syntax Notation Number One"). If you wonder why there is a full stop in the acronym – there's a simple reason for it. First written as "ASN1" many people just misread it as ASNI (note the letter "I" and not the digit "1" at the end). And this rapidly mutated into ANSI, what was probably not meant at all. So the OSI wrote it with a full stop and no-one every confused it with the ANSI anymore.

## 8.2 Get Ready for the Acronyms

If you don't care where the rules come from, or what they are for, you can skip this section.

I happen to work in the telecommunications industry, and I have experience with ASN.1 and related standards as defined by the ITU (others know these standards either through the ISO or from RFC's). ASN.1 is used by the GDMO (roughly, "Guidelines for the Development of Managed Objects" – there are several ways I know of to decompose that acronym) to describe MIBs (Management Information Bases). The "Simple Network Management Protocol" (SNMP) for example works with a MIB describing various aspects of a networked device (like network cards, routing tables, and IP addresses).

Basically, ASN.1 is a way to describe data types in a machine independent way from a text description (something like XDR – the eXternal Data Representation used by ONC/RPC. Go and find out yourself what the latter is). Vaguely associated with ASN.1 are sets of encoding rules, such as BER (or Basic Encoding Rules) which describe how to actually create external representations of data. There are other encoding rules (e.g., FER), but you've had enough acronyms for now. ASN.1 is really a very powerful tool – you may want to learn more about it on your own.

## 8.3 How It Works

Ok, enough of the background. Let's see how it works in practice. The basic idea is to describe data elements as a three piece combination: tag/length/value, sometimes referred to as TLV. You basically have:

- a tag, which describes what type of data this is,
- a length, which says how long the following value is,
- and a value, which is basically an octet (or byte) sequence that describes the value.

The basic unit of information is the octet (or byte): 8 bits of information. You can see how 8 bits might be a little small to describe large strings – more on that later. One thing that must be noted is that TLVs can be nested, that is, the value part of a TLV tuple can contain TLVs.

I'm going to skip a full description of BER and just report the basics of how they relate to `XmStrings`. Let's take a trivial example:

```
xmstr = XmStringCreateLtoR("Hello\nWorld", XmFONTLIST_DEFAULT_TAG);
```

The first thing to notice is the `XmFONTLIST_DEFAULT_TAG`. That's a clue to M∗TIF that the string passed in is represented in the current locale (I'm not even going to try to talk about NLS – look elsewhere for what locale means). The second thing to notice is that we used `XmStringCreate-LtoR`, which means the function should be aware of separators (normally, this means "look for newlines"). So M∗TIF would parse that as "`Hello`" (locale text), "`\n`" (separator in this locale), and "`World`" (locale text).

| Identifier | Value |
|---|---|
| XmSTRING_COMPONENT_UNKNOWN | 0x00 |
| XmSTRING_COMPONENT_CHARSET | 0x01 |
| XmSTRING_COMPONENT_TEXT | 0x02 |
| XmSTRING_COMPONENT_DIRECTION | 0x03 |
| XmSTRING_COMPONENT_SEPARATOR | 0x04 |
| XmSTRING_COMPONENT_LOCALE_TEXT | 0x05 |

**Table 8.1:** *Component identifiers for `XmStrings`.*

Let's look at what Motif does tell us about encodings – each `XmString` component has a different identifier (see figure 8.1). Hmm, these could be the tag part of the TLVs! Given that, the `XmString` that M∗TIF 1.2 generates is the following (in hex and chars, with the `0x` prefix removed from the hex):

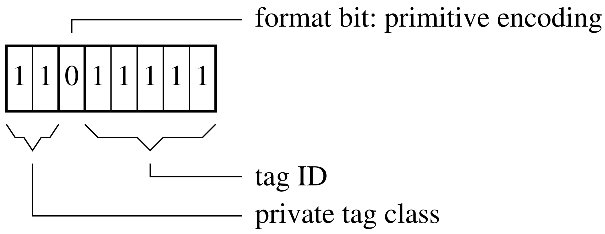
```
DF 80 06 10 05 05 'H' 'e' 'l' 'l' 'o' 04 00 05 05 'W' 'o' 'r' 'l' 'd'
```

which makes absolutely no sense when you look at it that way. Try this:

| | |
|---|---|
| `0xDF 0x80` | this is a M∗TIF string (essentially) |
|   `0x06 0x10` |   which contains a 16 byte `XmString` |
|     `0x05 0x05` |     which contains 5 bytes of locale text |
|       "`Hello`" |       which has the value "`Hello`" |
|     `0x04 0x00` |     and a separator |
|       – *nothing* – |       which has no data (never does) |
|     `0x05 0x05` |     and 5 more bytes of locale text |
|       "`World`" |       which has the value "`World`" |

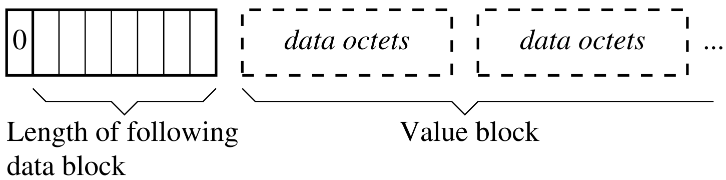
The first number (on lines that have them) is the tag; the second number is the length. You can see that this description shows how TLVs can be nested. Look at it this way; if I just describe the string above structurally, it comes out as (using parentheses as an indicator of nesting): TLV=(TLV=(TLV,TLV,TLV)).

The first tag value `0xDF` identifies every `XmString`. While this value seems arbitrary at the first glance it makes some sense. The tag value can be decomposed into three separate fields as shown below.

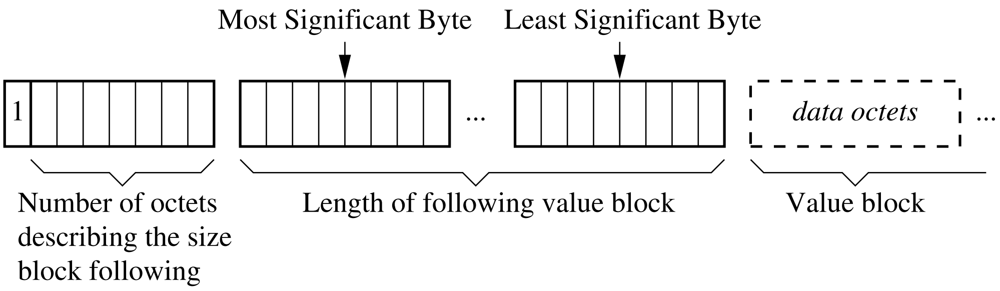



The most significant bits 7 and 6 indicate that this is a private tag class, thus the bits 4 to 0 are just set to an arbitrary value. The “F” flag (bit 5) indicates that this is a simple tag encoding and not a composed one. Ok, after this you’re scratching your head once again. Where does the next value `0x80` (the first length) fit in? Remember how I said that 8 bits was a little small for describing lengths? Well, that’s where BER kicks in. There are really three ways for describing lengths: short form, long form, and indeterminate form. As far as I know, Motif cheats horribly on this (more on this below). Here’s how you describe lengths in BER:

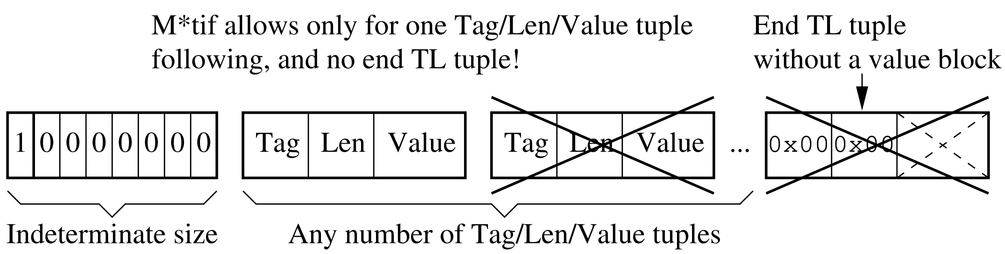
- If the length $<$ `0x80`, then length is contained in one octet.




- If the length $>$ `0x80` (but not indeterminate), then the length octet is defined as `0x80` plus the number of octets needed to describe the length (up to 127 additional octets, so this can describe lengths up to $2^{127 \cdot 8}$, or $2^{1016}$, which is *really* huge). The octets describing the length immediately follow the length octet and come before the value octets. In practice (as far as I know), M*TIF limits this to two additional length octets, which implies a maximum value length of 65535. Maybe the CSF once planned to port M*TIF to M$Windooze…




- If the length > $2^{1016}$, or you are really lazy (like M*TIF is), then the length octect contains `0x80`, and you're to parse the value (which contains TLV tuples) until you come to a TLV whose tag and length are both 0. M*TIF uses the indeterminate form only for encoding the first TLV, but never when encoding the subsequent TLV's.




As I said before, M*TIF is really lazy (what else did you expect?!). The first header (`0xDF 0x80`) should imply that an `XmString` parser should look for a tag and length that are both 0. In practice, Motif strings contain only *one* element in the value: the `XmString`. I've parsed strings in M*TIF looking for the (`0x00 0x00`) tag/length, and run off into space. Therefore, LESSTIF stops after finding the first `XmString` component. In effect, a length of `0x80` in M*TIF means "I don't know how long my value is, but my value is really a TLV, and there's only one of them".

Let's look at our example string again, in light of this information:

| | |
|---|---|
| `0xDF 0x80` | `XmSTRING_TAG`, `XmSTRING_LENGTH` |
|   `0x06 0x10` |   `XmSTRING_COMPONENT_XMSTRING`, 16 bytes |
|     `0x05 0x05` |     `XmSTRING_COMPONENT_LOCALE_TEXT`, 5 bytes |
|       "`Hello`" |       "`Hello`" |
|     `0x04 0x00` |     `XmSTRING_COMPONENT_SEPARATOR`, 0 bytes |
|       *– nothing –* | |
|     `0x05 0x05` |     `XmSTRING_COMPONENT_LOCALE_TEXT`, 5 bytes |
|       "`World`" |       "`World`" |

That should make more sense, now. Note that the tags 6–125 are said to be reserved in M*TIF's header files; now you should understand why the value 6 is `XmSTRING_COMPONENT_XMSTRING` (which doesn't appear in any Motif header). The length `XmSTRING_LENGTH` is used within LESSTIF as a synonym for the indeterminate length value of `0x80`. You'll find its definition in `$(LESSTIF_ROOT)/libXm/XmString.c`.

## 8.4 Structures

[Need to explain here why order is important in the strings – the charsets MUST come before the strings that use them].

## 8.5 The Other Side of XmStrings

As you can easily imagine, the ASN.1 representation of a `XmString` isn't very suitable for fast handling, yet saves memory. Another advantage of a `XmString` is that it is independent of a `XmFontList`. Only when you need to know the height and/or width of a `XmString` or want to render it into a drawable a `XmFontList` must be specified so that the individual string components can be "connected" to the fonts from the font list.

If you need to work many times with a particular `XmString` (like the list widget) it is more convenient to "compile" the ASN.1 representation of a `XmString` into an internal form – a `_XmString`. The individual TLV's from the ASN.1 `XmString` are thereby transformed into string components accessible through pointers. The compilation is carried out with `_XmStringCreate()` which takes a `XmString` and returns a `_XmString`. Next after the transformation the references to the fonts should be resolved – use `_XmStringUpdate()` for this task. Lateron, you can update the fonts the (internal) `_XmString` will use whenever you want by calling `_XmStringUpdate()`.

A `_XmString` is merely a pointer to a `__XmStringRec`. This structure points to a table of pointers to the string's components. In addition the `__XmStringRec` also accounts for the size of that table of pointers (see figure 8.1). As the table of pointers can grow and shrink whenever the string gets manipulated, it may also move in memory. Thus the the `_XmString` pointer can't point directly to the components table but must point to a data structure instead which stays at the same memory location all the time (well – at the same *logical* or *linear* memory location as modern virtual memory management may move memory blocks at any time around the physical memory).

In turn the string components are described by `__XmStringComponentRecs`, which contain the type of a component (see table 8.1 on page 106), the component's data, its length, and finally the font to be used. Every `__XmStringComponentRec` can thus be regarded as a TLV converted to a more suitable form to the CPU. The `font` member of the component record is just an index into a `XmFontList`. So be sure to update these indices with `_XmStringUpdate()` whenever the font list changes which is used for rendering the `_XmString`.

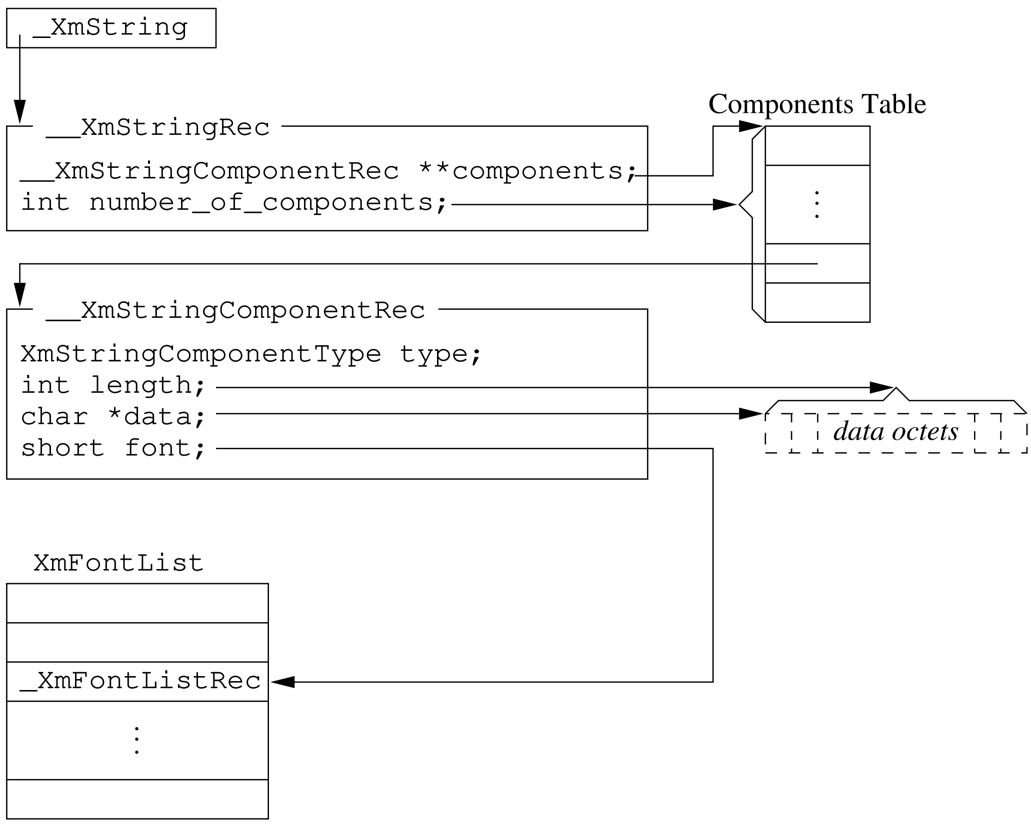



**Figure 8.1:** *Internal representation of a "compiled"* `_XmString`.

# 9

# Hash & Cache

Harald Albrecht

## 9.1 Introduction

Caches within LESSTIF serve two main purposes: avoiding unnecessary round-trips to the X server as well as resource sharing. The resource sharing can either occur on the server side (pixmaps, graphics contexts,...) or on the side of the client (images, memory). Caching can also improve performance (although this is at some times only an idle wish...).

When working with caches, very often you need to check a cache for the existence of a particular item. Because the cache may contain many items this lookup has to be fast. In almost every case a simple linked list isn't suitable when you need speed – but (at least) a linked list is easy to code.

As a way out the `XContexts` of the Xlib come to mind. Unfortunately, they can only be used if you have a display pointer ready at hand. And a `XContext` is destroyed when the display it belongs to is closed. Therefore the `XContexts` are neither suitable for all caching purposes nor as a general associative array.

## 9.2 The Hash Table Module

Whenever you need a cache or an associative array that must be independent of a particular display pointer then you should use the generic hash table mechanism within LESSTIF. Generally, it offers much better performance than a simple linked list, especially if there are many items to manage. The hash table mechanism makes reinventing the wheel unnecessary in almost every case.

You can think of a hash table as some kind of associative array. You first put a value named by an identifier into the hash table. Lateron you can ask for the value using the identifier. Both an identifier and its associated value make up an "item". The values and identifiers for the hash tables within LESSTIF are typedef'ed in a portable fashion:

```
typedef XtPointer LTHashItemID;
typedef XtPointer LTHashItemValue;
```

This allows you to use the broad range of integral data types in C for both values and identifiers. And if the space provided by these data types isn't sufficient, you can use the identifiers and values as pointers to structures instead. Because of this broad range of data types you may have to provide functions for comparing two items and calculating a hash value of an item:

```
typedef unsigned int (*LTHashGetHashFunction)(LTHashItemID);
typedef Boolean (*LTHashCompareFunction)(LTHashItemID, LTHashItemID);
```

A `LTHashGetHashFunction` returns an unsigned integer that represents the hash value for that particular item specified as the parameter to the function. A `LTHashCompareFunction` must return `True` if the two items specified by the parameters are equal.

Hash tables are created and destroyed using the following two functions. You don't have to supply a size when creating a hash table because LESSTIF's hash tables grow as needed whenever new entries are added.

```
LTHashTable LTHashTableCreate(LTHashGetHashFunction GetHash,
                              LTHashCompareFunction Compare,
                              unsigned int IDSize);
```

Creates a fresh hash table and returns a pointer to it for subsequent use. You can either specify your own functions for calculating a hash key and comparing items (more precisely: comparing their identifiers) in `GetHash` and `Compare` or `NULL`. In the latter case the hash table will use default functions. If you use data structures as identifiers then you probable have to supply your own functions.

The final parameter `IDSize` indicates what type of identifiers you're working with and whether the memory occupied by the identifier belongs to the hash table. If you specify here `LTHASH_ID_NOCOPY`, then the hash table will not make a copy of a data structure an identifier points to when adding or replacing items. Another special case are strings for which you can specify `LTHASH_ID_STRING`. The hash table will then take care of copying and freeing the string identifiers. If you specify for `IDSize` any size (other than zero or one, as these ones are reserved), then the hash table mechanism will copy the data structure of that size pointed to by a `LTHashItemID` to private allocated storage whenever you add or replace items in the hash table.

```
void LTHashTableDelete(LTHashTable ht);
```

Deletes a hash table and frees all memory occupied by it.

At any time you can ask a hash table how much items it currently contains.

```
int LTHashTableGetNumItems(LTHashTable ht);
```

After creating a hash table you can add (or remove) items to (from) it.

```
Boolean LTHashTableAddItem(LTHashTable ht, LTHashItemID id,
                           LTHashItemValue value);
```

Adds the item identified by `id` with the value `value` to the hash table. If there is already an item with the same identifier in the hash table then the function doesn't modify the value of that item and returns `False`. Otherwise, the function adds the item to the table and indicates success by returning `True`.

If either `id` or `value` are pointers to data structures, make sure that these data structues are not allocated in automatic storage. Because the hash table only stores the pointers you must not free the data structures until the item is removed from the hash table. The only exception occurs when you have specified the size of the data structure of your item identifiers when creating the hash table. In this case the hash table will make a copy of the identifier.

```
Boolean LTHashTableReplaceItem(LTHashTable ht,
                               LTHashItemID id,
                               LTHashItemValue value,
                               LTHashItemValue *value_ret);
```

Much the same as `LTHashTableAddItem()`. But when the hash table already contains an item with the identifier `id`, then `LTHashTableReplaceItem()` replaces the item's value with `value`. The item's previous value is returned in `*value_ret` as long as you don't specify a `NULL` pointer for the final parameter. If a replace took place then the function returns `True`. This return value is especially useful if the value of an item is a pointer to memory allocated using `XtMalloc()`. In this case you can free the memory occupied by the old value whenever `LTHashTableReplaceItem()` returns `True`.

```
Boolean LTHashTableReplaceItemAndID(LTHashTable ht,
                                    LTHashItemID id,
                                    LTHashItemValue value,
                                    LTHashItemID *id_ret,
                                    LTHashItemValue *value_ret);
```

Much the same as `LTHashTableReplaceItem` but this time even the identifier of the item will be replaced with the new identifier. If you don't specify `NULL` for `id_ret` then you'll get the old identifier of the item. In the cases where the hash table mechanism takes care of the memory occupied by identifiers (within the hash table), it'll free that storage if the identifier of an item was replaced.

```
Boolean LTHashTableRemoveItem(LTHashTable ht, LTHashItemID id,
                              LTHashItemID *id_ret,
                              LTHashItemValue *value_ret);
```

Removes an item identified by `id` from the hash table and returns `True` if it succeeds In this case you will get back the identifier and the value of the item in `*id_ret` and `*value_ret` as long as you don't specify `NULL` for these pointers. The information returned can help you to free the data allocated to hold the identifier and the value. In the cases where the hash table mechanism takes care of the memory occupied by identifiers (within the hash table), it'll free the storage used by the identifier.

```
Boolean LTHashTableLookupItem(LTHashTable ht, LTHashItemID id,
                              LTHashItemValue *value);
```

Looks up an item identified by `id` within the hash table `ht`. If it succeeds then it returns the value of the item in `value` and returns `True`. Otherwise the function returns `False` if the item can't be found.

```
int LTHashTableForEachItem(LTHashTable ht,
                           LTHashForEachFunction iter,
                           XtPointer ClientData);
```

From time to time you need to iterate over the contents of a hash table. This is where you'll use this iterator function. For every item in the hash table the iterator function `iter` is called. The function prototype for such an iterator function is as follows:

```
typedef LTHashForEachIteratorResult
    (*LTHashForEachFunction)(LTHashTable,
                             LTHashItemID, LTHashItemValue, XtPointer);
```

The iterator gets as its final parameter the `ClientData` parameter from the call to `LT-HashTableForEachItem`. The iterator function then should return one of the following results depending on whether the iteration process should continue or not. The function `LTHashTableForEachItem` returns the value of a counter that is initialized at the start

| Identifier | Operation |
| --- | --- |
| LTHASH_BREAK | Exit the iteration loop. |
| LTHASH_CONT | Continue. |
| LTHASH_COUNT | Continue and increment the counter. |
| LTHASH_COUNTANDBREAK | Increment the counter but exit the iteration loop. |

**Table 9.1:** *Results the iterator function of a hash table can return.*

of the iteration process and is incremented whenever the iterator function indicates this.

# A

## Appendix

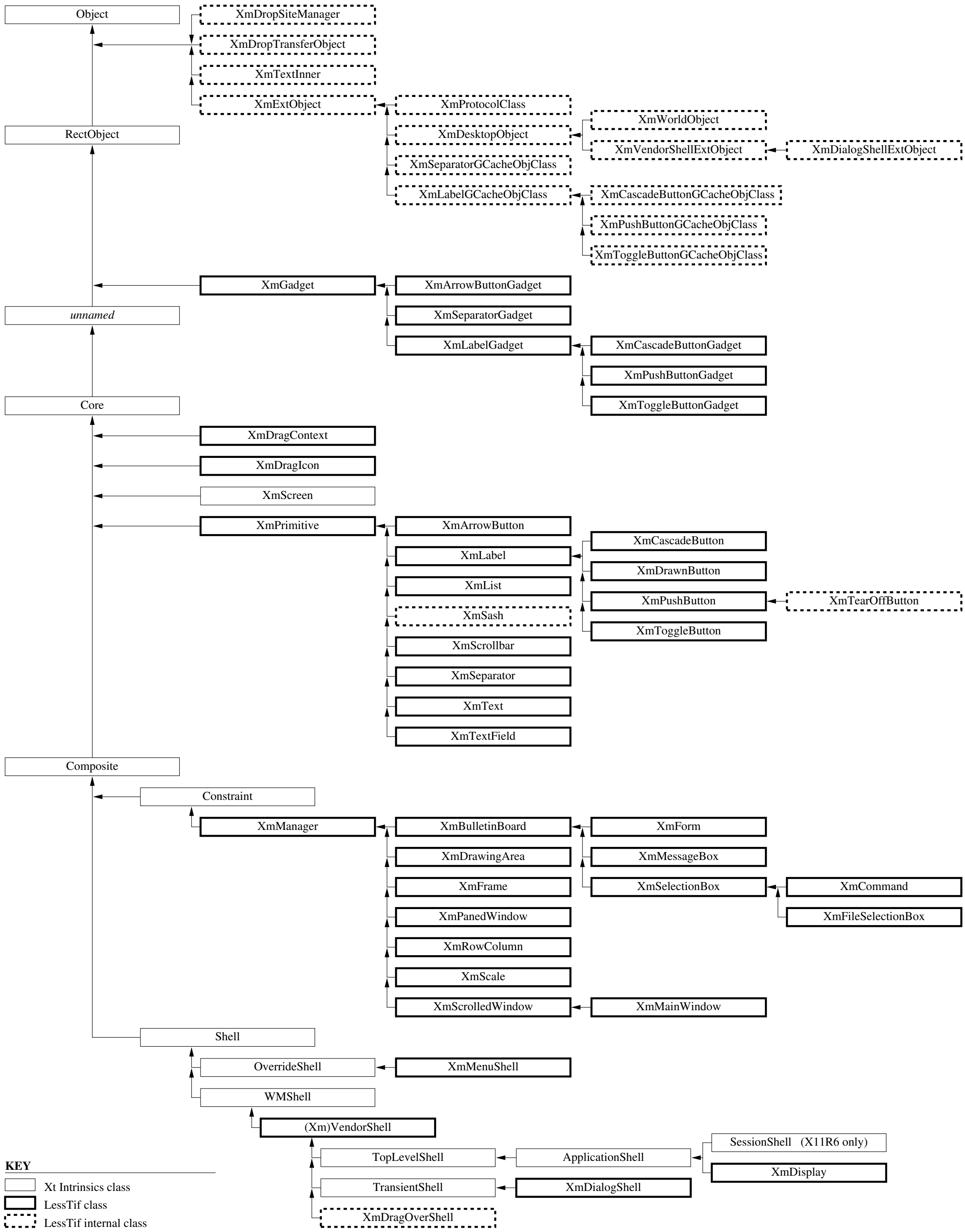



**Figure A.1:** *The big picture of all widget classes.*

# Index

## K



## L



## M



## O



## P



## Q



## R



## S



## T



## V



## W



## X