BX^6

Customizing Builder Xcessory

ICS

Integrated Computer Solutions Incorporated

Integrated Computer Solutions, Inc.

54B Middlesex Turnpike, Bedford, MA 01730

Tel: 617.621.0060

Fax: 617.621.9555

E-mail: info@ics.com

WWW: http://www.ics.com

Third printing

January 2009

Contents

How to Use This Manual

Overview

The following table provides an overview of *Customizing Builder Xcessory*:

Chapter 1—Extending Builder Xcessory	Introduces basic concepts about how to customize and extend Builder Xcessory.
Chapter 2—Adding Widgets	Describes how to add widgets to Builder Xcessory.
Chapter 3—Adding C++ Components	Describes how to add class components to Builder Xcessory.
Chapter 4—Adding Resource Type Editors	Describes how to add resource type editors to Builder Xcessory.
Chapter 5—Adding Predefined Callbacks	Describes predefined callbacks and how to add them to Builder Xcessory.
Chapter 6—Builder Xcessory Functions	Describes functions you can use to customize Builder Xcessory.
Chapter 7—Using the BX Object Packager	Describes the components of and how to use the Builder Xcessory Object Packager.
Chapter 8—Modifying the WML File	Describes the components of and how to modify OSF/Motif Widget Meta Language (WML) files, as well as how Builder Xcessory uses WML files.
Chapter 9—Creating Other Control Files	Describes the different control files Builder Xcessory uses, and how to create them.
Chapter 10—Using Custom Objects	Describes how to incorporate your own widgets and C++ components into Builder Xcessory, and provides a feature checklist of widget functionality.

Notation Conventions

This document uses the following notation conventions:

{BX}

The syntax `{BX}` refers to the directory into which the Builder Xcessory is installed. The default directory is the following:

```
/opt/bxpro-6.0
```

Index

Most index entries are in lowercase:

fonts
    fixed width 28
    non-XLFD 228

Entries capitalized in Builder Xcessory are capitalized in the index:

Force To Selected 161
Force to Selected Tree 161

Languages

Because Builder Xcessory supports multiple programming languages, not all explanations or examples are applicable to all languages. The following icons indicate sections specific to particular languages:

C	C	U	UIL	V	ViewKit	C++	C++	J	Java

Note: Information that applies to all Motif environments does not use icons. In text, Motif refers to C, C++, ViewKit, and UIL.

Lists

The following two types of lists present procedures:

1. Numbered lists present steps to perform in sequence.

- Bulleted lists present alternate methods.

Objects

Objects are indicated as follows:

- Palette collection names are capitalized words with intercaps:

  Form or PushButton
- Instance names are lowercase words with intercaps:

  form or pushButton
- Class names are capitalized words with intercaps:

  Test or MyClass

Menu Notation

To indicate menus and menu options, the following format is sometimes used:

BX_window_name : menu_name : menu_item(or dialog_selection)

For example, Browser:File:Exit is the Exit selection from the File menu of the Browser window.

Text

- Literals such as file names, directory paths, code and text as typed on the command line are printed in `Courier` font:

  ```
  .bxrc
  /usr/ics
  ```
- Text preceded by a % denotes text to enter at the command line:

  ```
  % bx
  ```
- Book titles, chapter names, and section names appear in *italic*:

  Builder Xcessory Reference Manual

  “Updating the Resource Editor” on page 136
- The main windows of the Builder Xcessory are capitalized as follows:

  Main Window

  Resource Editor

Definitions

This document uses the following terms:

Click	Move the cursor over an object, press a mouse button, and immediately release the mouse button. When the button is unspecified, assume MB1 (typically the left mouse button).
Collection	A group of related user interface objects saved to the Builder Xcessory Palette for reuse. Collections can include any UI object supported by Builder Xcessory, including widgets, gadgets, C++ classes, or ViewKit components.
Component	A user interface object, generally used in the context of ViewKit classes. ViewKit components generally consist of collections of Motif widgets along with code to implement general or specific functionality, encapsulated into a C++ class subclassed from an element of the ViewKit application framework.
Cursor	A graphical image appearing on the screen which reacts to the movements of a mouse or other pointing device. In the Builder Xcessory, the cursor appears as an angle bracket when creating a widget, and an arrow when selecting a pull-down menu or menu item. During a drag and drop operation, it appears as an icon reflecting the type of object dragged and the target over which it is positioned.
Drag	Press a mouse button, then move the mouse without releasing the button. Typically followed with a drop operation. The phrase, “drag on to” indicates a drag and drop operation. Use MB2 to perform a drag and drop operation, unless otherwise specified.
Drop	Release the mouse button after positioning the mouse (and screen object) as desired. Typically follows a drag operation. The phrase, “drop on to” indicates a drag and drop operation. Use MB2 to perform a drag and drop operation, unless otherwise specified.
Enter	Type a new value and press the Enter key.
Gadget	A user interface object built upon the Xt Intrinsics (the X Toolkit). Similar to a widget, a gadget lacks certain data structures (such as a window) contained in a widget. The gadget maintains this data on the client side, rather than on the server, as does a widget. Although seldom used with today’s server performance levels, gadgets remain supported by BX.
{lang}	Specifies the currently selected language for code generation.

MB1, MB2 and MB3	Mouse buttons one, two, and three. Typically, MB1 is the left-most button, MB3, the right-most. On a two-button mouse, MB2 is most commonly emulated by pressing both buttons simultaneously. For actions described as "click," assume MB1.
MB3 Quick Access menu	This menu is invoked by pressing MB3 while the mouse pointer is positioned over an object on the display. The contents of the menu depend on the type of object pointed to and the window in which you access the menu.
Object/ UI object	A reusable software entity with a user interface (UI), or visible, aspect. A generic term for the various objects that are manipulated with Builder Xcessory. UI objects include widgets, related collections of widgets, C++ classes, and ViewKit components. The term object and the term UI object are interchangeable.
Paste buffer	A cache into which a cut or copied object is placed. Also called a cut buffer.
Resize	To change the height and/or width of an object.
Resource	A user preference specification that controls elements of an application that can be customized by the user.
Select	To choose an object to be acted upon or an action to be performed; accomplished by clicking mouse button one on an object or menu item.
Session	A single, continuous working session, from the time you start Builder Xcessory from the command line, or from another tool, to the time you select Exit from the Browser's File menu.
Widget	A user interface object built upon the Xt Intrinsics (the X Toolkit). Motif user interface objects are widgets.
WML	OSF/Motif Widget Meta Language (WML).

Prerequisite Knowledge

This document assumes that you are familiar with the X Window System and OSF/Motif. If you are developing with ViewKit objects, this document assumes that you are familiar with these toolkits. Consult the following documentation lists for recommended references.

OSF/Motif documentation

For detailed descriptions of OSF/Motif and X, refer to the following documentation:

- *Motif™ 2.1 Programmers Reference*. The Open Group, 1997. (ISBN 1-85912-119-5)
- *CDE 2.1/Motif™ 2.1™ Style Guide*. The Open Group, 1997. (ISBN 1-85912-104-7)

X Window System documentation

- Asente, Paul, Donna Converse, and Ralph Swick. *X Window System Toolkit*. Digital Press, 1997. (ISBN 1-55558-178-1)
- Scheifler, Robert W. and James Gettys. *X Window System-Core Library and Standards*. Digital Press, 1996. (ISBN 1-55558-154-4)
- Scheifler, Robert W. and James Gettys. *X Window System-Extension Libraries*. Digital Press, 1997. (ISBN 1-55558-146-3)
- Scheifler, Robert W. and James Gettys. *X Window System-Core and Extension Protocols*. Digital Press, 1997. (ISBN 1-55558-148-X)

CDE documentation

For information about the Common Desktop Environment (CDE) widgets, refer to the following documents:

- *CDE 1.0 Programmer's Guide*. Addison-Wesley, 1995. (ISBN 0-201-48954-6)
- *CDE 1.0 User's Guide*. Addison-Wesley, 1995. (ISBN 0-201-48951-1)

ViewKit documentation

For a description of ViewKit (VKit) classes, refer to the following documentation:

- *ViewKit ObjectPak™1.5/2.1 Programmer's Guide*. Integrated Computer Solutions, 2002. (Included with the purchase of BX PRO.*)*
- *IRIS ViewKit™ Programmer's Guide*. Silicon Graphics, Inc. 1994. (Document Number 007-2124-002)

EPak documentation

For a description of EnhancementPak (EPak) widgets, refer to the following Integrated Computer Solution's documentation (included with BX PRO):

- *EnhancementPak™ 3.0 Programmer's Reference*. Integrated Computer Solutions, 2002.
- *GraphPak™ Programmer's Reference.* Integrated Computer Solutions,2002.

Note: If you are using BX PRO, you can use and compile the EnhancementPak widgets and ViewKit objects in your interface. If you are using Builder Xcessory, you can use the EnhancementPak widgets and ViewKit objects in your interface, but you must purchase their respective libraries to compile any interface built with the EnhancementPak widgets or ViewKit objects. Contact your Sales Representative for more information.

Extending Builder Xcessory

1

Overview

This chapter includes the following sections:

- **Extending Builder Xcessory**
- **Summary of Customization Procedures**

Extending Builder Xcessory

You can customize and extend Builder Xcessory to handle new widgets and component classes, different resource editors, and your own callback functions. This capacity for extensibility makes Builder Xcessory a premier part of your development environment, one that you can use with all objects you are working with.

Steps for customizing Builder Xcessory

Extending Builder Xcessory involves the following two steps:

- Ensuring that the extra data is available to the Builder Xcessory binary.
- Telling Builder Xcessory how to handle that data.

Ensuring Availability of Data to Builder Xcessory

Types of data include the following functions that Builder Xcessory can call:

- Widget class data describing new widgets
- C++ classes describing new components.

Regardless of the type of data, making the data available to the Builder Xcessory binary is accomplished in the same way. On most systems, you can make a shared library of the classes or functions and place it in a location that Builder Xcessory searches when the library is needed, so that the library is dynamically loaded.

Using an object file

As of Builder Xcessory 6.0, the shared library mechanism is supported on all systems supported by Builder Xcessory 6.0. If you do not have access to a shared version of your library, you can link with object file `bx.o` to produce a new Builder Xcessory binary.

Telling Builder Xcessory How to Handle Data

You can tell Builder Xcessory how to handle the data in several ways, depending on the type of customization. In general, however, the following two primary mechanisms are used in combination:

- Providing extra files that Builder Xcessory reads on start-up.
- Providing functions that Builder Xcessory calls, and having those functions make special calls to Builder Xcessory to handle the customization.

On most systems these functions and calls can be in a library that Builder Xcessory loads when it first needs to call the function.

Builder Xcessory Object Packager

For certain kinds of customization, such as adding new widgets or components, or editing a catalog control file, Builder Xcessory provides the Builder Xcessory Object Packager tool. You can use the Object Packager to make these customizations whether or not you are using a system that handles dynamically loading shared libraries. The Object Packager helps you set up the control files that Builder Xcessory requires on all systems. For more detailed information, refer to *Chapter 7—Using the BX Object Packager*.

Summary of Customization Procedures

This document examines the various customization procedures in detail. The following table summarizes how customizations are accomplished:

Type of Customization	**Where Data Goes**	**Tell Builder Xcessory How To Use Data**
New widgets	In a library, loaded dynamically from the shared library path or linked with the `bx.o` file.	Use a WML file and other control files.
New ViewKit or UI components	In a library, loaded dynamically from the shared library path.	Use a WML file and other control files.
New Resource Editors	In a library, loaded dynamically from certain Builder Xcessory areas or linked with the `bx.o` file.	Use extra functions linked to the library making calls to extend Builder Xcessory.
New Callback Functions	In a library, loaded dynamically from certain Builder Xcessory areas or linked with the `bx.o` file.	Use extra functions linked to the library making calls to extend Builder Xcessory.

Adding Widgets 2

Overview

This chapter includes the following sections:

- **Obtaining a Widget**
- **Making the Widgets Available**
- **Generating WML And Other Control Files**

Obtaining a Widget

Adding widgets[1] to Builder Xcessory is a very simple procedure. The following sections describe how to add widgets to Builder Xcessory.

User-defined Widgets

You can add your own X11R5/Motif 2.1 Xt Intrinsics-based widgets to Builder Xcessory. These “user-defined” widgets appear on the Palette and can be accessed and manipulated just like any other Palette collection. You must specify the widget you wish to add to Builder Xcessory by a C file conforming to the standard guidelines for widget writing. For more information about this style, refer to Asente, Converse, and Swick’s *X Window System Toolkit*.

Information Sources

Widgets are available from a number of commercial sources, as well as freely available from numerous locations on the Internet. The following Usenet newsgroups and website are good sources of information about both commercial and free widgets:

- `http://www.motifzone.net`
- `comp.windows.x.motif`
- `comp.windows.x`

XmDumbLabel widget files

For the remainder of this document, we will use the XmDumbLabel widget as an example. XmDumbLabel is a simple Motif widget that displays an XmString with resources for fonts, color, and margins. Source code for the XmDumbLabel widget can be found in the file `{BX}/xcessory/examples/AddWidget.c`. The XmDumbLabel widget source code consists of the following files:

File	Description
`XmDumbLabel.c`	Source code for the widget.
`XmDumbLabel.h`	Public header file for the widget.
`XmDumbLabelP.h`	Private header file for the widget.

If you want to add more than one widget, you can group them together into one library.

1. In this chapter, the term “widget” encompasses widgets and also gadgets.

Making the Widgets Available

On most systems, Builder Xcessory accesses widgets by loading a shared library when the widget class is first accessed. To load widgets dynamically using Builder Xcessory, the following two requirements must be met:

- Widgets must be compiled into a shared library.

  On most systems, you can do this by compiling the various objects of the library with the `position-independent` code flag and linking the library with any other libraries on which the new library depends. Consult your system and compiler manuals for exact details on building shared libraries.

- Widgets must have a Motif-style creation function that returns an unmanaged instance of the widget.

  A Motif-style creation function has the following function prototype:

```
Widget CreateFunction(Widget parent, char *name,
                      ArgList args, Cardinal num_args)
```

How Builder Xcessory Searches for a Library

The first time you create an instance of a widget in a shared library, Builder Xcessory searches for the library in several locations in addition to the usual shared library search path. Once it finds the library, Builder Xcessory loads the library and finds and calls the widget's creation function.

Complete search path

The complete search path is as follows

```
${HOME}/.builderXcessory6/lib
{BX}/xcessory/lib
{BX}/lib
system shared-library search path
```

Note: On most systems, the shared-library search path is an environment variable such as `LD_LIBRARY_PATH or SHLIB_PATH`. Consult your system for details.

The name of the shared library to load is specified in the WML file for the widget. Each widget specifies a shared library to find and load in the LoadLibrary WML attribute. For more information on the WML file format, see *"Generating WML And Other Control Files"* on page 12 and *Chapter 8—Modifying the WML File*.

Specifying the Widget Creation Function

On most systems, Builder Xcessory dynamically loads a shared library containing the widget and searches the WML file for the function to use to create it. Set the CreationFunction attribute to name the function to call to create the widget. The function has the same signature as the Xm-style convenience functions described in the section *"Creation function"*. In addition, set the ConvenienceFunction attribute to name the function that the code generator should use in generated code (in most cases, these two functions will have the same value[1]).

Creation function

The creation function is required for dynamically-loaded widgets. Most widgets written for use with the Motif widget set provide such a function, in the Motif style. For example, ICS EnhancementPak widgets have routines with names such as XiCreateButtonBox.

If the widget you are integrating does not provide such a function, you can easily create one, as shown in the following example:

```
#include <My/Widget.h>

Widget CreateMyDumbLabel(Widget parent, String name,
   ArgList args, Cardinal ac)
{
   return XtCreateWidget(name, xmDumbLabelWidgetClass,
   parent, args, ac);
}
```

Compiling the creation function

To compile the creation function, use the following procedure:

1. Use the correct position-independent code flag for your compiler.
2. Create a shared library that contains this new object and is linked to the necessary widget library.
3. Put this new intermediary library in one of the directories that Builder Xcessory searches.
4. Specify the new function in the widget's creation function WML attribute.

Adding Widgets Using the `bx.o` File

On systems where using shared libraries is not feasible, the widget must be linked into the Builder Xcessory binary, along with a control function that makes a call to extend Builder Xcessory. When you extend Builder Xcessory, you must provide the following three functions, which Builder Xcessory calls at start-up:

1. Some advanced Builder Xcessory users provide their own creation function for specific processing in addition to creating the widget, but name the usual ConvenienceFunction so that generated code is generated correctly.

```
void AddUserWidgets()
void AddUserDefinedEditors()
void AddUserFunctions()
```

In our example, these functions are in a file called `addWidget.c`. To create a new version of Builder Xcessory that contains your new widgets, relink `addWidget.o` with the object file `bx.o` and the libraries containing the widgets to be integrated.

`AddUserWidgets` is an entry point for Builder Xcessory to use to add new widgets. It takes no arguments and has no return value. If you rebuild Builder Xcessory for any reason, you must provide at least an empty version of `AddUserWidgets`.

Using AddUser Widgets

Tell Builder Xcessory about the availability of new widgets by making calls to `RegisterWidgetExtended` from the function `AddUserWidgets`. Call `RegisterWidgetExtended` once for each new widget you add to Builder Xcessory. `RegisterWidgetExtended` has the following function prototype:

```
void
RegisterWidgetExtended(char*class_name,WidgetClassRec
*class_ptr, char *conv_fct_name, XtPointer
conv_fct_ptr, char *include_file, char
*resource_prefix)
```

`class_name`	Name of widget class being added to Builder Xcessory. For example xmDumbLabelWidgetClass or xiButtonBoxWidgetClass.
`class_ptr`	Address of the class pointer. For example, the class of XmDumbLabel is xmDumbLabelWidgetClass. Entered as `&xmDumbLabelWidgetClass`. If NULL, you must provide a convenience function to create the widget.
`conv_fct_name`	Name of convenience function to create widget. The value should be set to NULL if there is no special convenience function to create the widget. If this value is NULL, Builder Xcessory uses XtCreateWidget to create an instance of the widget when it generates code.
`conv_fct_ptr`	Pointer to function named by `conv_fct_name`. Depending on whether or not `class_ptr` is NULL, the convenience function should be Motif-style or Xt-style (see *“Xt-style creation routine”* on page 11). If you provide a function here, Builder Xcessory uses it to create the widget internally.
`include_file`	File that must be included in order to reference the

	`class_ptr` and any other functions or data defined by the widget. For example, XmDumbLabel requires you to include "`<Xm/XmDumbLabel.h>`".[1] If the widget requires more than one include file, separate each file in the string by a space, such as "`<Xm/Foo.h> <Xi/Bar.h>`". In the unlikely event that the widget does not require an include file, `include_file` can be NULL.
`resource_prefix`	Widget set prefix used by the widget for its resource names. For example, Motif uses "Xm" as its resource prefix. The Athena widget set uses "Xt". If you specify NULL for this value, "Xm" is assumed.

Note: When adding multiple widgets, the `RegisterWidgetExtended` calls must be made in SuperClass to SubClass order. For example, if you had a widget XmDumbPushButton that was sub-classed from widget XmDumbLabel, you would first call `RegisterWidgetExtended` for XmDumbLabel and then call `RegisterWidgetExtended` for XmDumbPushButton.

1. The path assumes that the header file is installed with your other Motif header files.

Example

As an example of using `RegisterWidgetExtended`, we show the call used to add the XmDumbLabel widget to Builder Xcessory.

```
        void
        AddUserWidgets()
        {
      RegisterWidgetExtended("xmDumbLabelWidgetClass",
                                          &xmDumbLabelWidgetClass,
                                          NULL, NULL, //
                                          "<Xm/XmDumbLabel.h>, "Xm");
        }
```

Sets widget creation routine to default XtCreateWidget, both internally and in generated code.

The `RegisterWidgetExtended` call in the `AddUserWidgets` function allows you to override how widgets are created.

Motif-style creation routine

Builder Xcessory supports two types of creation function. The Motif-style creation routine has the same function prototype as any Motif XmCreate function:

```
Widget CreateFunction(Widget parent, char *name,
  ArgList args, Cardinal num_args)
```

Xt-style creation routine

The Xt-style creation routine has the same function prototype as the XtCreateWidget function:

```
Widget CreateFunction(char *name, WidgetClass widget_class,
 Widget parent, ArgList args, Cardinal num_args)
```

When you register an alternate creation routine, you can choose between either of the two styles. Also, you can choose whether the routine is used internally by Builder Xcessory and/or externally in the code generated by Builder Xcessory.

Using an Xt-style creation routine

To use an Xt-style creation routine, you must provide a value for `class_ptr` in `RegisterWidgetExtended`. How you specify values for `conv_fct_name` and `conv_fct_ptr` determines where the new creation routine is used.

If you set `conv_fct_name` to NULL, Builder Xcessory uses `XtCreateWidget` to create the widget in the code that is generated. If `conv_fct_name` is not set to NULL, Builder Xcessory uses the string you specified as the name of the function to call to create an instance of the widget.

If you set `conv_fct_ptr` to NULL, Builder Xcessory uses `XtCreateWidget` to create the widget internally. If `conv_fct_ptr` is not set to NULL, Builder Xcessory uses the function you specified to create an instance of the widget.

Using a Motif-style creation routine

If you set `class_ptr` to NULL in `RegisterWidgetExtended`, Builder Xcessory assumes a Motif-style creation routine. You must then provide a value for `conv_fct_ptr`. Otherwise, Builder Xcessory cannot create an instance of the widget.

If you set `conv_fct_name` to NULL, Builder Xcessory uses XtCreateWidget in the generated code and the function you supplied in `conv_fct_ptr` for creating widget instances internally.

Building a new Builder Xcessory executable

Once you create your Builder Xcessory interface file, you must rebuild the Builder Xcessory executable. An example `makefile` is available in `{BX}/xcessory/examples/RebuildBX/Makefile`.

The command line you use to recompile the Builder Xcessory executable varies from system to system. In general, use a command similar to the following:

```
cc -o bx addWidget.o [objects] {BX}/lib/bx.o
[libraries]
```

The example Makefile contains command lines for all of the platforms supported by Builder Xcessory 6.0.

Generating WML And Other Control Files

Control files

Your new widgets are now available to Builder Xcessory. Builder Xcessory uses the following control files to know what to do with the widgets:

- Widget Meta Language (WML) file

  Describes the widget’s position in the class hierarchy as well as all the resources used by the widget.
- Catalog file
- Pixmap Icon file
- TCL files
- Collection files

Refer to *Chapter 8—Modifying the WML File* and to *Chapter 9—Creating Other Control Files* for more detailed information about control files.

Builder Xcessory Object Packager

The catalog and WML files are the primary control files. The Builder Xcessory Object Packager is a tool designed to help you create and modify these files. For more detailed information about the Builder Xcessory Object Packager, refer to *Chapter 7—Using the BX Object Packager*. You can use the Builder Xcessory Object Packager on all systems. On most systems, the Object Packager can read the shared library containing your widgets. You can set the Object Packager to determine the resources for those widgets and write them out to the WML file. Refer to *Chapter 8—Modifying the WML File* for more detailed information.

Modifying WML Files

ICS has extended the base syntax for WML (Widget Meta-Language, described on page 1-1142 of the *OSF/Motif Programmer's Reference* for Release 1.2) to allow its use with Builder Xcessory. Refer to *Chapter 8—Modifying the WML File* for more detailed information.

Because not all widgets are written to be manipulated by an interface builder, you must test all resources added by the new widget to make sure they behave as expected. Much of a widget's behavior in Builder Xcessory is customizable through Builder Xcessory's extensions to WML. Test each resource, decide what to change about its behavior, and make the appropriate changes to the Builder Xcessory-produced WML file.

Adding C++ Components

3

Overview

This chapter includes the following sections:

- **Adding Components**
- **Creating a Component (CreationFunction)**
- **Managing Subclasses of Existing Components**
- **Editing a Component (AttributeFunction)**
- **Editing Resources On Subclass Components**
- **Components That Can Take Children**

Adding Components

The procedure for adding new C++ components, such as ViewKit classes or UI class components, is similar to the procedure for adding widgets. The classes are in a shared library, within the same search path. Several control files define Builder Xcessory's use of the classes.

The primary difference between the procedure for adding class components and the procedure for adding widgets is in the interface that Builder Xcessory uses to manipulate the class components. Builder Xcessory can use standard functions to manipulate all widget and gadgets because they are subclasses of the basic Xt widget classes. The mechanisms for manipulating widgets and gadgets are defined by Xt. For example, there are standard Xt functions to create widgets and to change their resource values. For Class Components, no such standards are available.

Because there is no standard way to manipulate ViewKit or UI class components, you must provide standardized interface functions such that Builder Xcessory can operate on the components. Your code then acts as a filter between Builder Xcessory and the component you are operating on.

Create a shared library that contains these functions (and possibly the component as well) and place it in a directory in the library search path used by Builder Xcessory. The names of the functions are specified in the WML specification for each component. Whenever you create an instance of the component in question, Builder Xcessory loads the shared library and uses the functions you specified in the WML specification.[1]

The following sections describe five functions you can use to manipulate class components.

Creating a Component (CreationFunction)

Regular instantiable components

An instantiable component is a component that can be created directly with a call to `new`. This routine can have any name, but must match the Xm-style creation prototype:

```
void* CreateComponent(Widget parent, char* name,
   ArgList args, Cardinal arg_count);
```

This routine should create an instance of your component and return the instance as a `void*` cast. Your component must be a subclass of UIComponent

1. Builder Xcessory calls all these functions, including widget classes. Advanced Builder Xcessory users might want to monitor what Builder Xcessory is doing with a widget class; they can provide these other functions. Normally, however, the functions are required only when integrating class components.

(generated by Builder Xcessory) or VkComponent. The following example illustrates a sample create method:

```
extern "C" void*
CreateMyComponent(Widget parent, char* name,
   ArgList args, Cardinal ac)
{
   MyComponent* obj = new MyComponent(name, parent);
   XtSetValues(obj->baseWidget(), args, ac);

   return (void*)obj;
```

Conditions for calling the routine

This routine is called under the following conditions:

- When a new instance of the component is required.
- When the component needs to be recreated, for whatever reason.

Abstract Components

An abstract component is a component that cannot be created directly with a call to `new`. The user of the class must first make a subclass of this class before instantiation with `new` can take place. Builder Xcessory cannot create instances of abstract classes so the integration of these classes is somewhat more involved than the regular instantiable component class shown in the previous section.

Integrating an abstract class

The first step in integrating an abstract class is to create a class for which Builder Xcessory can create an instance. This class must implement all abstract methods provided by the abstract base class. We recommend prefixing your class name with “BX” to distinguish this class as a Builder Xcessory integration class. If you are integrating an abstract class called MyAbstract, then your integration class can be called BXMyAbstract (this name is not enforced by Builder Xcessory but it is a convenient naming convention).

The create routine now creates an instance of the new subclass:

```
extern "C" void*
CreateMyAbstract (Widget parent, char* name,
    ArgList args, Cardinal ac)
{
    BXMyAbstract* obj = new BXMyAbstract (name, parent);
    EditMyAbstract ((void*)obj, True, args, ac);

    return (void*)obj;
}
```

Important: All actions performed on this component are performed on your subclass, although in Builder Xcessory it appears as though they are applied to the abstract class.

Builder Xcessory must also enforce that whenever the user creates an instance of this abstract class, a concrete subclass is first created within Builder Xcessory, which is then instantiated for the user. To inform Builder Xcessory that a class should be automatically subclassed whenever an instance is created, use the WML flag:

```
AutoSubclass = "MyAbstract";
```

The quoted string is the name of the class being subclassed. In most cases this string will be the same as the abstract class name.

When creating any abstract class it is common to give protected scope to methods that alter the state of the abstract portion of the component. This ensures that the subclass can manipulate its behavior without having to have those methods publicly available.

When creating a concrete subclass for Builder Xcessory to instantiate, those protected methods are not available to the Edit function described in *“Editing a Component (AttributeFunction)”* on page 24. To facilitate editing these resources (by calling the protected methods), the Builder Xcessory subclass should make all these methods public. By default, they call the abstract classes protected methods.

Managing additional constructor parameters

Because most components are composed of several widgets or components, it is unlikely that your component will have resources applied only to its base widget. As far as the user of your component is concerned, there is a set of resources presented to them in the Builder Xcessory Resource Editor.

Methods For Setting Resources

The component writer can set these resources using one of the following methods:

- Setting the method on the class

- Setting the constructor parameter

Setting methods on a class

Refer to *“Editing a Component (AttributeFunction)”* on page 24 for more detailed information on resources set using a set method on the class.

Setting the constructor parameter

The only time in the integration process that you actually create an instance of an object is in the creation function. Therefore, you must manage constructor parameters within the creation function.

Note: Do not create an instance of the object in any function but the creation function.

To allow additional constructor parameters, scan the ArgList passed to the Create method for any resources that correspond to constructor parameters. The appropriate constructor can then be called based on which resources are passed.

You can recreate a component more than once with Builder Xcessory. If you have resources that are constructor parameters, any change to these attributes in Builder Xcessory causes the component to be recreated. Builder Xcessory passes all resources that have changed from their default values to the creation function. To mark a resource as a constructor parameter, add the following to the Resource definition in the WML file:

```
Recreate = True;
```

It is highly likely that constructor parameters have no corresponding get method associated with them. If this is the case, then the resource should also be specified as:

```
NeverVerify = True;
```

Example create method dealing with constructors

The following example illustrates a create method that deals with additional constructor parameters:

```
extern "C" void*
CreateMyComponent(Widget parent, char *name,
        ArgList args, Cardinal ac)
{
    // Storage for the constructor parameter value.

    char *label = NULL;


    // So as not to pass the constructor parameter resource
    // to the Edit function, we'll allocate a new resource
    // list and copy in only the resources not corresponding
    // to constructor parameters.

    Cardinalrsc_count = 0;
    ArgList resources=(ArgList)XtMalloc(ac*sizeof(Arg));


    // Now scan the list for constructor parameters.

    for (int i = 0; i < ac; i++)
         if (!strcmp("label", args[i].name))
        {
            label = (char *)args[i].value;
        }
        else
        {
            XtSetArg(resources[rsc_count],
                     args[i].name, args[i].value);
            rsc_count++;
        }
    }

    // Create the component with the constructor
    //parameter and pass the remaining attributes to the
    //Edit function.

    MyComponent*obj = new MyComponent(name, parent, label);
        EditMyComponent((void *)obj, True, resources, rsc_count);


        // Free the allocated memory.

        XtFree((char *)resources);


        // Return the new component.

        return (void *)obj;
}
```

In this example, the resource “label”[1] was removed from the ArgList before the

ArgList was passed on to the Edit function. Removing the resource is a wise move, because this resource might be applied to the base widget of the component, producing unwanted results (as shown in *"Editing a Component (AttributeFunction)"* on page 24).

Managing Subclasses of Existing Components

While most components integrated into Builder Xcessory are relatively primitive, the component model that Builder Xcessory follows encourages the creation of abstract classes and subclasses of existing classes. If a class's superclass has already been integrated into Builder Xcessory, then some of the work described in *Chapter 8—Modifying the WML File* will be reduced.

Subclasses

Subclasses present more of an issue for editing the component's resources. *"Editing Resources On Subclass Components"* on page 27 describes regular resources that have set/get methods provided in the sub- or superclass. However, as described in*"Managing additional constructor parameters"* on page 18, some resources are used as constructor parameters, which poses a difficult problem. The CreateComponent integration method for the superclass already deals with these resources. But, because that method creates an instance of our superclass, it is a practical impossibility to call the superclass's CreateComponent method in a subclass.

Writing a new CreateComponent method

One solution to this problem is to write a new CreateComponent method for the subclass that contains all the code required to deal with the superclass's constructor resources. This results in duplication of code for the superclass's constructor resources. If the superclass is changed then all the subclass integration code will need to be modified.

Writing a routine

An alternative solution to this problem is to write a routine that is called from the creation function for both the superclass and subclass. This routine can have any prototype, but at the very least will need to take an ArgList (containing a list of resources) and a Cardinal (set to the number of items in the ArgList). This routine would scan the resource list for constructor parameters and somehow provide the values to the caller. Both the superclass and subclass creation function could then use this routine to scan for the constructor parameter and so save recoding this search for every subclass that needs it.

1. Although a literal string is used here, usually the value is defined in a header file (as with widget resources). It might appear as XzNlabel or MyNlabel.

Example

Taking this approach, the creation function used for MyComponent in the previous example is split into two functions as shown in the following example.

```
char *
  FindMyComponentLabel(ArgList before, Cardinal before_count,
            ArgList after, Cardinal *after_count)
  {
     // Storage for the constructor parameter value.

     char*label = NULL;

     // Scan the list for constructor parameters.

     for (int i = 0; i < before_count; i++)
     {
        if (!strcmp("label", args[i].name))
        {
           label = (char *)args[i].value;
        }
        else
        {
           XtSetArg(after[*after_count],
                 before[i].name, before[i].value);
           *after_count++;
        }
     }
     return label;
  }

  extern "C" void *
  CreateMyComponent(Widget parent, char *name,
               ArgList args, Cardinal ac)
  {    // So as not to pass the constructor parameter resource
       // to the Edit function, we'll allocate a new resource
       // list and copy in only the resources not corresponding
       // to constructor parameters.

       Cardinal  rsc_count = 0;
       ArgList   resources = (ArgList)XtMalloc(ac*sizeof(Arg))

       // Extract the constructor parameter from the ArgList

       char*label = FindMyComponentLabel(args, ac,
                        resources, &rsc_count);

       // Create the component with the constructor parameter
```

```
        // and pass the remaining attributes to the Edit function.

        MyComponent*obj = new MyComponent(name, parent, label);
        EditMyComponent((void *)obj, True, resources, rsc_count);

        // Free the allocated memory.

        XtFree((char *)resources);

        // Return the new component.

        return (void *)obj;
    }
```

Now, if we created a subclass of MyComponent that also needed to extract the `label` constructor parameter, its creation function would also call FindMyComponentLabel().

Editing a Component (AttributeFunction)

Editing component resources

Typically, you edit object attributes in C++ with set/get method pairs. Usually, classes supply only a set method. Rarely does the component supply only a get method. Builder Xcessory allows you to provide a method to do the binding between ArgList items and methods on a component by providing a single method to accomplish both set and get functions. This function should check each Arg in the ArgList to see if it matches a component resource (which will be defined in the WML file), or a base widget resource. The function supplied should match the prototype:

```
void EditComponent(void* object, Boolean set, ArgList args,
Cardinal arg_count);
```

Example Edit Method

This function is called every time an attribute is changed in the Builder Xcessory Resource Editor. The following section illustrates a sample edit method:

```
extern "C" void
EditMyComponent(void *object, Boolean set,
        ArgList p_args, Cardinal p_ac)
{
    MyComponent *obj = (MyComponent *)object;

    // A loop index variable.

    int i;

    // Allocate two argument lists --
    // one for base widget resources, used when values
    // are both set and retrieved.

    ArgList bw_args = (ArgList)XtMalloc(p_ac * sizeof(Arg));
    Cardinal bw_ac = 0;

    // and one for component level resources, used ONLY
    // when values are retrieved.

    ArgList c_args = (ArgList)XtMalloc(p_ac * sizeof(Arg));
    Cardinalc_ac = 0;

    // Loop through the parameter argument list either set or
    // get the values appropriately.

    for (i = 0; i < p_ac; i++)
    {
        if (!strcmp("myAttribute", p_args[i].name))
        {
          if (set)
          {
```

```
          // p_args[i].value contains a pointer to
          // the value to set.

          MyAttributeType*val =
              (MyAttributeType *) p_args[i].value;

          // Set the value of "myAttribute" by calling
          // the corresponding method.

          obj->setMyAttribute(val)
        }
            else
            {
                // Get the current value of "myAttribute"
                // and store it in c_args[c_ac].value.

                XtSetArg(c_args[c_ac], p_args[i].name,
                            obj->getMyAttribute());
                c_ac++;
            }
        }
        else
        {
        // The attribute doesn't correspond to a component
        // method, so keep it to set or get on the base
        //widget.

          XtSetArg(bw_args[bw_ac], p_args[i].name,
                p_args[i].value);
          bw_ac++;
        }
    }
    // Now take care of the base widget resources.

    if (set)
    {
        XtSetValues(obj->baseWidget(), bw_args, bw_ac);
    }
    else
    {
        XtGetValues(obj->baseWidget(), bw_args, aw_ac);
        // When we get values, we need to merge all the
        // retrieved values back into the p_args ArgList.
        // First put back the values fetched from the
        // base widget.

        for (i = 0; i < bw_ac; i++)
        {
            XtSetArg(p_args[i], bw_args[i].name,
                        bw_args[i].value);
        }
        // Then put back the values retrieved using
        // methods of the component. Remember to start in
        // the list where we left off in the last loop.

        for (i = 0; i < c_ac; i++)
        {
            XtSetArg(p_args[bw_ac + i], c_args[i].name,
                    args[i].value);
        }
    }
    // Free the allocated memory.

    XtFree((char *)bw_args);
    XtFree((char *)c_args);
    }
```

This function deals with one component attribute called "myAttribute". This resource will be described in the WML as a resource in standard resource format. The resource name in the WML code could be XmNmyAttribute.

Note: Builder Xcessory strips off the prefix “XmN”, and allows any prefix that you want to use before the first N. For example, “BcN,” “DtN,” “XiN,” and so forth.

This function performs the following tasks:

- Takes in a list of arguments in the standard Xt Arg structure and cycles through them
- Builds two destination lists one for component resources (c_args), and one for base widget resources (bw_args).

This routine behaves differently in the set and get modes. The two states are defined in the following sections.

Set mode

Note: In this case c_args is not actually used.

The argument “set” is True. For each Arg passed into the function in `p_args`, see if its name matches any of the component attributes (using `strcmp` to compare `Arg.name` to the attribute name). If the comparison succeeds, call the appropriate set methods to set the attribute using the value supplied in `Arg.value`. If all comparisons fail, add this Arg to the list of Args for the base widget.

Once each Arg has been checked, all that remains in the `bw_args` structure are resources to be applied to the base widget, so call XtSetValues and pass the `bw_args` structure.

Get mode

The argument “set” is False. In this case, we retrieve values from the component, so both `bw_args` and `c_args` are needed.

For each Arg passed into the function in `p_args`, see if its name matches any of the component attributes. If the comparison succeeds, call the appropriate get method to retrieve the value and assign that value to the `Arg.value`.

Some component attributes may not have a set/get pair. In this case, ignore the Arg and do not assign any value to `Arg.value`, or increment the counter `c_ac`. It is also a good idea to use the WML construct `NeverVerify = True` for this resource, so that Builder Xcessory never passes it to the Edit function.

If the comparison with all component attributes fails, assign the `Arg.name` to the `bw_args`. Once all component attributes have been retrieved, the `bw_args` array will contain those resources to be retrieved from the component's base widget. Call XtGetValues to get the values for these resources.

Now `c_args` contains all the attributes from the component, and `bw_args` contains all the attributes from the base widget. These are now recombined into `p_args` to be returned to Builder Xcessory.

This code can act as template for any EditComponent function.

Editing Resources On Subclass Components

The superclass integration files already deal with all the resources for the superclass and its base widget, so we can leverage that code. The EditComponent function for the subclass should deal only with its own resources, then pass any remaining resources up to the superclass's EditComponent integration method. This is simple for the set version of the function. The get version requires work similar to dealing with the base widget of a class in *"Editing component resources"* on page 24.

Set Mode

For the subclass, we must check for resources specific to this class, but only those not already dealt with by the superclass, or any of its superclasses. These resources should be removed from the ArgList and the remaining ArgList passed up to the superclasses EditComponent.

Example

The following code sample below shows only the set portion of the code required for MySubComponent integration:

```
extern "C" void
EditMySubComponent(void *object, Boolean set,
                ArgList p_args, Cardinal p_ac)
{
    MySubComponent *obj = (MySubComponent *)object;

    // A loop index variable.

    int       i;
    if (set)
    {
        // Allocate an argument list for superclass attributes

        ArgList s_args ArgList)XtMalloc(p_ac * sizeof(Arg));
        Cardinal   s_ac = 0;
        for (i = 0; i < p_ac; i++)
        {
            if (!strcmp("MySubAttribute", p_args[i].name))
            {
                // This is a subclass attribute, so set it.

                MySubAttributeType*val =
                    (MySubAttributeType *)p_args[i].value;
                obj->setMySubAttribute(val);
            }
            .
            .
            .
            else
            {
                // No subclass attribute applied to this, so
                // save it for passing to the superclass.

                XtSetArg(s_args[s_ac],
                        p_args[i].name, p_args[i].value);
                s_ac++;
            }
        }
        // Now call the superclass's Edit function.

        EditMyComponent(object, set, s_args, s_ac);
        // Free the allocated memory.

        XtFree((char *)s_args);
    }
}
```

Get Mode

This adds the complication that it is necessary to find the subclass's values and pass any others to the superclass. Once the superclass returns its values, they must be merged with the subclass's before they can be returned.

Code

The following code example merges the get code into the previous example:

```
extern "C" void
EditMySubComponent(void *object, Boolean set,
                   ArgList p_args, Cardinal p_ac)
{
    MySubComponent *obj = (MySubComponent *)object;

    // A loop index variable.

    int i;

    // Allocate an argument list for superclass attributes

    ArgList s_args=(ArgList)XtMalloc(p_ac * sizeof(Arg));
    Cardinal   s_ac = 0;

    // Allocate an argument list for values retrieved via
    // methods of this component.

    ArgList c_args=ArgList)XtMalloc(p_ac * sizeof(Arg));
    Cardinal   c_ac = 0;
    for (i = 0; i < p_ac; i++)
    {
        if (!strcmp("MySubAttribute", p_args[i].name))
        {
            if (set)
            {
            MySubAttributeType *val =
                (MySubAttributeType *)p_args[i].value;
            obj->setMySubAttribute(val);
            }
            else
            {
                XtSetArg(c_args[c_ac], p_args[i].name,
                        obj->getMySubAttribute());
                c_ac++;
            }
        }
        .
        .
        .
        else
        {
            // No subclass attribute corresponds to this
            // attribute name, so pass it up to the superclass.

            XtSetArg(s_args[s_ac], p_args[i].name,
                    p_args[i].value);
            s_ac++;
         }
      }
  // Now call the superclass's Edit function.

    EditMyComponent(object, set, s_args, s_ac);

    // Merge the retrieved values back into p_args

    if (!set)
    {
        // First merge in the values retrieved
        // from the superclass.

        for (i = 0; i < s_ac; i++)
        {
            XtSetArg(p_args[i],
                    s_args[i].name, s_args[i].value);
        }
        // Then merge in the values retrieved
        // from component methods.

        for (i = 0; i < c_ac; i++)
        {
            XtSetArg(p_args[s_ac + i], c_args[i].name,
                    c_args[i].value);
        }
    }
    // Free the allocated memory.

    XtFree((char *)s_args);
    XtFree((char *)c_args);
  }
```

Components That Can Take Children

If the component can take children, the following three functions might be required:

- ChildParentFunction
- ChildFunction
- ConstraintFunction

The following sections describe these functions in detail.

Obtaining the Parent for Children (ChildParentFunction)

To identify the parent of a given widget, use the ChildParentFunction:

```
Widget GetWidgetParent(void* object);
```

This function retrieves the ID of the widget to use as the XtParent of any child widgets or components added to this component. If this routine is not provided, Builder Xcessory uses the base widget of the component.

```
extern "C" Widget
GetMyComponentWidgetParent(void* object)
{
        MyComponent* obj = (MyComponent*)object;
        return obj->getAddParent();
}
```

Adding A Child To The Component (ChildFunction)

If the component needs to do any internal record keeping after a new child has been added, you can specify a ChildFunction to be called:

```
    void ModifyChildList(void* object, Widget child,
                                             Boolean add);
```

This routine is used to notify the component that a child has been added or is about to be removed, allowing the component to initialize or clean up any internal data relating to the child. An example of this is a component like a VkTabbedDeck, which may have an add method to register a widget or component as a tab panel after the child has been created:

```
extern "C" void
ModifyMyComponentChildList(void* object, Widget child,
                                Boolean add)
{
        MyComponent* obj = (MyComponent*)object;

        if (add)
        {
            obj->addTabPanel(child);
        }
        else
        {
            obj->removeTabPanel(child);
        }
}
```

Editing Child Constraint Resources (ConstraintFunction)

If the component has attributes that can be set for each child, you can provide a ConstraintFunction:

```
void EditComponentConstraint(void* object, Widget child,
                    Boolean set, ArgList args, Cardinal ac);
```

This routine provides for the case of a component applying constraint style attributes to its children, like the TabbedDeck attribute tabLabel. The WML definition for these resources occurs in the section for the component, but the resource definition is declared as Constraint rather than Argument.

A resource declared in this way appears in the Constraint Resources section of any children added to MyComponent. When those resources on the children are changed, it is this method, `EditComponentConstraint()`, that gets called to set the value. This function is nearly identical to the component Edit function, except that both the component pointer and the child widget are passed as parameters.

```
extern "C" void
EditMyComponentConstraint(void *object, Widget child,
            Boolean set, ArgList args, Cardinal ac)
{
    MyComponent*obj = (MyComponent *)object;
    for (int i = 0; i < ac; i++)
    {
        // This is a constraint attribute
        // defined by MyComponent.
        if (!strcmp(args[i].name, "childLabel"))
        {
            if (set)
            {
                obj->setChildLabel(child,
                            (char *)args[i].value);
            }
            else
            {
                *((char **)(args[i].value)) =
                  obj->getChildLabel(child);
            }
        }
    }
}
```

Adding Resource Type Editors

4

Overview

This chapter includes the following sections:

- **Adding Resource Type Editors**
- **Creation Functions**
- **Update Functions**
- **Fetch Functions**
- **Registering Resource Type Editors**

Adding Resource Type Editors

Adding extended editors

Builder Xcessory understands and provides editors for a large number of resource types. From time to time, you will add a widget or class component to Builder Xcessory that defines a new resource type. In order to more easily work with the new resource type, you might decide to add a new resource type editor (an extended editor) to Builder Xcessory.

Note: The procedure for adding new resource type editors is similar to the procedure for adding widgets.

On most systems, Builder Xcessory dynamically loads the shared library containing the callbacks. If the function AddUserDefinedEditors is in the callback library, that function is called. In the AddUserDefinedEditors function, you can make calls to other functions to register the callbacks with Builder Xcessory on all systems.

Building the library

To establish the library, build a shared library and include an object file with the function AddUserDefinedEditors defined in the file.

When Builder Xcessory starts, it searches the following locations (such that definitions in the second override definitions in the first):

```
{BX}/xcessory/lib/editors
{HOME}/.builderXcessory6/lib/editors/
```

Builder Xcessory opens all shared libraries, and searches for the function AddUserDefinedEditors. Builder Xcessory calls all AddUserDefinedEditors functions it finds.

Static Integration

On systems where shared libraries are not feasible, you must modify your interface file (`addWidgets.c` in our example), and add to the AddUserDefinedEditors functions defined in the file. Then, re-link Builder Xcessory with `addWidgets.c` and with a new file containing the callbacks, as described in *Chapter 2—Adding Widgets*.

Example

For example, the XmDumbLabel defines a new resource type named “justify”. We will add both a simple type editor (the editor that appears in the Resource Editor) and an extended editor (the editor that appears when you click on the “...” button next to the simple editor).

Modifying the WML file

With respect to XmDumbLabel, modification of the WML file generated by Builder Xcessory is unnecessary. However, Builder Xcessory cannot recognize that the XmNjustify resource is of a special type.

Modify your WML file to include a DataType entry for the justify data type as follows:

```
DataType justify {
     TypeName = “Justify”;
     TypeSize = sizeofChar;
};
```

Changing the resource definition

Change the resource definition for XmNjustify to the following resource:

```
XmNjustify:Argument{
     Type=justify;
};
```

The next time you run Builder Xcessory, the XmNjustify resource for the XmDumbLabel widget will use the editors specified for the `justify` datatype. For a more detailed description of the options available in the WML file, refer to *Chapter 8—Modifying the WML File*.

Entry points

Builder Xcessory views a resource type editor as a “black box” with only three known entry points:

- Creation
- Update (Display)
- Fetch

Defining functions

We define both resource type editors for the justify resource type according the following information:

- Two creation functions (one for the simple editor and one for the extended editor)
- Two update functions
- Two fetch functions

The following sections describe these functions in detail.

Creation Functions

Builder Xcessory calls the creation function when it requires a new copy of the resource type editor. To implement the editor creation function, create the widgets that comprise your editor. Builder Xcessory creates any surrounding widgets (such as the XmDialogShell and the OK, Reset, and Dismiss buttons).

Simple editor

In the case of the simple editor, Builder Xcessory does impose some constraints on its size. Builder Xcessory passes any required resource values to the creation function in an Xt resource argument list.

Widget Hierarchy Generated in the Creation Function

The widget hierarchy you generate in the creation function should be a singly-rooted tree. That is, all widgets of your simple or extended resource type editors should descend from a single container.

Creation Function Prototype

The creation function has the following function prototype:

```
Widget EditorCreateFunc(Widget parent, ArgList pargs,
     Cardinal pc, XtPointer data)
```

`parent`	Parent widget passed to the creation function by Builder Xcessory. Create a container child (for example, a XmForm or XmRowColumn) from `parent` and build your editor widget hierarchy in that container.
`pargs`	Resource argument list for the widget resource that you apply to the container widget for your editor's hierarchy.
`pc`	Count of the widget resources applied to the container widget for your editor's hierarchy. If you plan to add any resources yourself, use the `XtMergeArgLists` function to combine your resource list with that provided by Builder Xcessory.
`data`	Pointer to generic data. You can use data in any way you want. Simply allocate memory and assign the pointer to data. The data pointer is passed to all of the other functions associated with this instance of the type editor.

Once the function finishes building the appropriate widget hierarchy, it returns the top-level container widget that it created.

To highlight the use of the resource type editor creation function, we'll examine the functions for the XmNjustify resource of the XmDumbLabel widget. Source code for all of the justify resource editor functions is in the file `{BX}/xcessory/examples/AddEditor.c`.

In this case, both the simple and extended editors build the same widget hierarchy. The only difference between the two editors is that the simple editor builds a horizontal radio box, but the extended editor builds a vertical radio box.

Simple Editor Creation Function

The following code is from the simple editor creation function:

```
Widget JustifySingleBuild(Widget parent, ArgList pargs,
                    Cardinal pc, XtPointer data)
{
```

... Variable Declarations...

Sets memory allocation in data as well as the variable ‘wids’. BuildCommonData does the memory allocation and then assigns the memory to `*(JustifyWidgets**)data`.

```
   JustifyWidgets *wids = BuildCommonData(data);
```

... Variable Declarations...

Shows how to set your own resources and merge them with the resources provided by Builder Xcessory.

```
   n = 0;
   XtSetArg(args[n],XmNorientation,XmHORIZONTAL);n++;
   margs = XtMergeArgLists(parg, pc, args, n);
   radio = XmCreateRadioBox(parent,"justifyExtBox",margs,
                             pc + n);
```

...Create toggle button gadgets...

Because the editor has no OK/Cancel buttons, we must provide BX with a callback for triggering an update of internal data.

```
   SetRscEditorUpdate(wids->sngl_left,XmNdisarmCallback);
   SetRscEditorUpdate(wids->sngl_cntr,XmNdisarmCallback);
   SetRscEditorUpdate(wids->sngl_right,XmNdisarmCallback);
      XtManageChild(radio);
      return(radio);
```

This example also shows the use of the simple utility function, `SetRscEditorUpdate`. When you supply your own Resource Editor entry, Builder Xcessory does not create an automated OK/Cancel button pair to confirm resource settings. One problem with this is that Builder Xcessory updates various internal data when you click OK.

Allowing Builder Xcessory to Update Internal Structures

SetRscEditorUpdate

To allow Builder Xcessory to update its internal structures, you must set a callback for Builder Xcessory to use. Use the function call SetRscEditorUpdate to set a callback for Builder Xcessory to use.

Only use this utility function in the code to create a simple resource type editor. SetRscEditorUpdate is not necessary in an extended editor.

SetRscEditorUpdate function prototype

SetRscEditorUpdate has the following function prototype:

```
void SetRscEditorUpdate(Widget wgt, char *callback_name)
```

`wgt`	Widget whose callback signals Builder Xcessory to update its internal data. For example, if you provide your own OK button, use that button as the `wgt` parameter.
`callback_name`	Name of the callback that should be used to signal Builder Xcessory. Builder Xcessory sets a callback function on `wgt` using the callback list specified by `callback_name`.

Update Functions

The update function is called by Builder Xcessory to display a value in a resource type editor. The value can be passed as either a string (the most common case) or as a value of the actual type. The string value used by Builder Xcessory is the same value that the fetch function returns.

Note: For Builder Xcessory to use a new resource type correctly, there must be a resource converter to convert a string representation of the resource to the actual expected type. Most widgets provide this type of converter for new types to allow the resource to be set from an application defaults file. See your Xt documentation for more information about resource converters.

EditorUpdateFunc Function Prototype

The update function has the following function prototype:

```
void EditorUpdateFunc(char *val_str, XtPointer val_ptr,
                  Widget wgt, XtPointer data,
                  XtPointer bx_internal)
```

`val_str`	String representation of the value of the resource that should be displayed in the resource type editor. This is the most common way Builder Xcessory passes a value to be displayed. Always try to use this value first when determining what to display. This value can be NULL.
`val_ptr`	Pointer to the value to display in the resource type. You must cast this pointer to the correct type. This method of passing a value to display is used rarely by Builder Xcessory. In most cases, this value is NULL.
`wgt`	Widget that is above the top-level container of your resource type editor. For an extended editor, `wgt` is the shell widget containing the editor. For a simple editor, `wgt` is the parent widget of your top-level container.
`data`	Pointer value that you supplied in the creation function. If you did not specify a value, `data` is NULL.
`bx_internal`	Used by Builder Xcessory. Do not modify this value.

Example

As an example of the resource type editor update function, we'll examine the functions for the XmNjustify resource of the XmDumbLabel widget:

```
void JustifySingleUpdate(String strval, XtPointer typeval,
                     Widget editor, XtPointer data,
                     XtPointer unused)
{
    JustifyWidgets *wids = (JustifyWidgets*) data;
    int        value = (int)typeval;
```

First, the string value is checked for a valid value. If it does not contain a valid string, the value containing a typeval is used.

```
   if ( strval )
   {
      if ( !strcmp("left", strval) )
      {
         value = XmALIGNMENT_LEFT;
      }
      else if ( !strcmp("center", strval) )
   {
         value = XmALIGNMENT_CENTER;
      }
      else if ( !strcmp("right", strval) )
      {
            value = XmALIGNMENT_RIGHT;
      }

   switch(value)
   {
   case XmALIGNMENT_LEFT:
      XmToggleButtonGadgetSetState(wids->sngl_left,
                                 True, False);
      XmToggleButtonGadgetSetState(wids->sngl_cntr,
                                 False, False);
      XmToggleButtonGadgetSetState(wids->sngl_right,
                                 False, False);
      break;
   case XmALIGNMENT_CENTER:
```

... Set Toggle Button Values ...

```
      break;
   case XmALIGNMENT_RIGHT:
      ... Set Toggle Button Values ...
      break;
   }
  }
```

Fetch Functions

The fetch function is called by Builder Xcessory when it wants to obtain the current value displayed in the resource type editor. The fetch function does what is necessary to convert the value currently being displayed to a string representation, and returns that value to Builder Xcessory. You can convert the returned string to the proper resource type by using the Xt resource conversion mechanism.

EditorFetch-Func function prototype

The fetch function has the following function prototype:

```
char * EditorFetchFunc(Widget wgt, XtPointer data)
```

wgt	Widget that is above the top-level container of your resource type editor. For an extended editor, `wgt` is the shell widget containing the editor. For a simple editor, `wgt` is the parent widget of your top-level container.
data	Pointer value that you supplied in the creation function. If you did not specify a value, `data` is NULL.

Registering Resource Type Editors

AddUserDefinedEditors is an entry point that Builder Xcessory uses to add new resource type editors. It takes no arguments and has no return value. If you are rebuilding Builder Xcessory using bx.o, you must provide at least an empty version of AddUserDefinedEditors.

Note: If you register an editor for a type already defined by Builder Xcessory (integer, font_list, etc.), your editor overrides the Builder Xcessory editor. This might be detrimental and is discouraged.

Use the function RegisterResourceEditor to provide Builder Xcessory with the list of functions to use to create, update, and retrieve values from the new resource type editor.

Call RegisterResourceEditor once for each new resource type editor that you are adding to Builder Xcessory.

RegisterResourceEditor Function Prototype

RegisterResourceEditor has the following function prototype:

```
void RegisterResourceEditor(char *resource_type,
                        EditorCreateFunc ext_create,
                        EditorUpdateFunc ext_update,
                        EditorGetFunc ext_fetch,
                        EditorCreateFunc simple_create,
                        EditorUpdateFunc simple_update,
                        EditorGetFunc simple_fetch)
```

resource_type	The name of resource type. In our example using the XmDumbLabel and its new resource "XmNjustify", the resource type is "justify".
ext_create	The resource type editor creation function for the extended editor. See the description of this function in *"Creation Functions"* on page 36. If this value is NULL, Builder Xcessory creates the default editor, which is a Motif Text widget.
ext_update	The resource type editor update function for the extended editor. See the description of this function in *"Update Functions"* on page 40. If this value is NULL, Builder Xcessory uses XmTextSetString to update the editor.
ext_fetch	The resource type editor fetch function for the extended editor. See the description of this function in *"Fetch Functions"* on page 42. If this value is NULL, Builder Xcessory uses XmTextGetString to retrieve a value.
simple_create	The resource type editor creation function for the simple editor. See the description of this function in *"Creation Functions"* on page 36. If this value is NULL, Builder Xcessory creates the default editor, which is a Motif Text widget.
simple_update	The resource type editor update function for the simple editor. See the description of this function in *"Update Functions"* on page 40. If this value is NULL, Builder Xcessory uses the XmTextSetString to update the editor.
simple_fetch	The resource type editor fetch function for the simple editor. See the description of this function in *"Fetch Functions"* on page 42. If this value is NULL, Builder Xcessory uses `XmTextGetString` to retrieve a value.

Example

As an example of using RegisterResourceEditor, the following code shows the call used to add the XmDumbLabel widget's "justify" type editor to Builder Xcessory:

```
void
```

```
AddUserDefinedEditors()
{
RegisterResourceEditor("justify",
   JustifyExtendedBuild,
   JustifyExtendedUpdate,
   JustifyExtendedFetch,
   JustifySingleBuild,
   JustifySingleUpdate,
   JustifySingleFetch);
}
```

Once you write the code for the functions specified in this example, you must add them to Builder Xcessory, as described in the following sections.

Compiling to a Shared Library

Adding functions to Builder Xcessory by compiling

The preferred method for adding functions to Builder Xcessory is to compile your new functions and function AddUserDefinedEditors() together into a shared library (refer to you platform documentation for information on how to build a shared library) and put this new library in a directory searched by Builder Xcessory:

```
${HOME}/.builderXcessory6/lib/editors
{BX}/xcessory/lib/editors
```

When you run Builder Xcessory next, it loads all libraries in these directories and calls AddUserDefinedEditors from each library.

Relinking Builder Xcessory

Adding functions to Builder Xcessory by relinking

An alternate method for adding functions to Builder Xcessory is to relink Builder Xcessory and provide function AddUserDefinedEditors (as well as empty implementations of AddUserWidgets and AddUserFunctions). Link everything with `bx.o`.

When you run the new Builder Xcessory executable, the new editors are available.

Adding Predefined Callbacks

5

Overview

This chapter includes the following sections:

- **Adding Callbacks**
- **Adding a Callback to Predefined Function List**

Adding Callbacks

Builder Xcessory allows you to add callbacks of your own design to the list of predefined callbacks in the Callback Editor. The procedure for adding predefined callbacks is similar to that of adding resource editors.

On most systems, Builder Xcessory dynamically loads the shared library containing the callbacks. If there is a function in that library called AddUserFunctions, it is called. In that function, you can make calls to other functions to register the callbacks with Builder Xcessory on all systems.

To construct this library, build a shared library and include an object file with the function AddUserFunctions defined in it.

When Builder Xcessory starts, it looks in both locations (the definitions in the second override the definitions in the first):

```
{BX}/xcessory/lib/functions
{HOME}/.builderXcessory6/lib/functions/
```

Static Integration

On systems where shared libraries are not available, you must modify your interface file (`addWidgets.c` in our example), and add to the AddUserFunctions defined in the file. Then, relink Builder Xcessory with `addWidgets.c` and with a new file containing the callbacks, as described in *Chapter 2—Adding Widgets*.

Adding a Callback to Predefined Function List

To add a callback function to the predefined function list, you must register the function by calling one of the following functions:

```
void RegisterUserCallback(char *name
                          XtCallbackProc fct,
                          char *unused)
void RegisterUserTypedCallback(char *name,
                          XtCallbackProc fct,
                          char *parameter_type)
```

name	The name of the function. You must provide a value for this parameter.
fct	A pointer to the callback function. If you do not provide a value for this parameter, Builder Xcessory cannot use the function in Play Mode.

`unused`	No longer used by Builder Xcessory.
	In previous versions of Builder Xcessory, the third argument to `RegisterUserCallback` was the name of a file containing the code to insert in the user's callbacks file when generating code for the named function. The file containing the code *must* now have the same name as the function. If this file cannot be found during code generation, Builder Xcessory produces only a stub function.
`parameter_type`	The name of the type for the client data parameter to the callback. If you specify a type name, Builder Xcessory allows the user to specify the function's parameter, as long as its type is the same as the type named by `parameter_type`.

Example

The following sections illustrate an example of adding the callback function InterceptWMDelete to the list of predefined callbacks in Builder Xcessory. InterceptWMDelete is written to be placed on a shell widget in its popupCallback.

InterceptWMDelete

InterceptWMDelete registers another function (Intercepted) with the X Translation Manager that is called when a shell receives the WM_DELETE_WINDOW protocol message.

The actual code for this function can be found in the `{BX}/examples` directory in the file `intwmdel.c`.

Adding the function to Builder Xcessory

To add this function to builder xcessory, you would include the following call to `RegisterUserCallback` in `AddUserFunctions`:

```
void AddUserFunctions()
{
    RegisterUserCallback("InterceptWMDelete",
        InterceptWMDelete,
        NULL);
}
```

Because `InterceptWMDelete` does not expect any client data, you might want to ensure that the user cannot enter a `client_data` parameter inside Builder Xcessory.

Ensuring that the user cannot enter client data

To ensure that the user cannot enter `client_data` parameters you must use RegisterUserTypedCallback and indicate a parameter_type of "None", as follows:

```
void AddUserFunctions()
{
    RegisterUserTypedCallback("InterceptWMDelete",
                InterceptWMDelete,
                "None");
}
```

This example registers the function InterceptWMDelete. When generating code, Builder Xcessory looks for a file with the same name as this function, InterceptWMDelete. If this file exists, Builder Xcessory inserts the code from this file in the Callbacks file as long as the function InterceptWMDelete is referred to and not already in the Callbacks file.

Builder Xcessory search order

Builder Xcessory searches for InterceptWMDelete in the `gen` directories. The order in which it searches is:

- `${HOME}/.builderXcessory6/gen/{LANG}`
- `{BX}/xcessory/gen/{LANG}`
- `{BX}/xcessory/gen/common`

where `{LANG}` is the language for which code is being generated. This search order allows you to better customize the generated code for the chosen language.

Builder Xcessory Functions

6

Overview

This chapter describes the following functions you can use to customize Builder Xcessory:

- **AddUserDefinedEditors**
- **AddUserFunctions**
- **RegisterUserCallback and RegisterUserTypedCallback**
- **RegisterUserEditor**
- **SetRscEditorUpdate**

RegisterUserCallback and RegisterUserTypedCallback

Use the function `RegisterUserCallback` or `RegisterUserTypedCallback` to add a predefined callback to Builder Xcessory.

```
void RegisterUserCallback(char *name,
    XtCallbackProc fct,
    char *unused)

void RegisterUserTypedCallback(char *name,
    XtCallbackProc fct,
    char *parameter_type)
```

AddUserDefinedEditors

Use the function `AddUserDefinedEditors` to add resource type editors.

```
void AddUserWidgets(void)
```

AddUserFunctions

Use the function `AddUserFunctions` to add predefined callbacks.

```
void AddUserFunctions(void)
```

RegisterUserEditor

Use the function `RegisterUserEditor` to register a resource type editor.

```
typedef void (*EditorCreateFunc)(Widget, ArgList
        Cardinal, XtPointer);
typedef void (*EditorUpdateFunc)(char *, XtPointer,
    Widget, XtPointer,
        XtPointer);
typedef void (*EditorFetchFunc)(Widget, XtPointer);
void RegisterResourceEditor(char *resource_type,
    EditorCreateFunc ext_create,
    EditorUpdateFunc ext_update,
    EditorGetFunc ext_fetch,
    EditorCreateFunc simple_create,
    EditorUpdateFunc simple_update,
    EditorGetFunc simple_fetch)
```

SetRscEditorUpdate

Use the function `SetRscEditorUpdate` in `simple_create` to tell Builder Xcessory when to update a widget with the current value.

```
void SetRscEditorUpdate(Widget wgt,
    char *callback_name)
```

Using the BX Object Packager

7

Overview

This chapter includes the following sections:

- **Builder Xcessory Object Packager**
- **Editing WML Files**
- **Background WML Files**
- **Editing the Catalog**
- **Unassigned Catalog**
- **Command-line Options and Resources**

Builder Xcessory Object Packager

The Builder Xcessory Object Packager is a tool for editing and writing WML and catalog control files. The WML and catalog files control most of the behavior of the classes in Builder Xcessory and how they appear on the Builder Xcessory Palette. In addition, the Builder Xcessory Object Packager can automatically generate the other necessary files (Tcl control and collection files).

The WML and catalog files are largely independent. The WML file describes various widget or component classes and how Builder Xcessory can manipulate them. The catalog file describes the classes that should appear on the Palette. It is possible to define a class in the WML file and have it not appear on the Palette. It is also possible to define an object to be generated in the catalog file while not (yet) supplying a definition in WML.

The Builder Xcessory Object Packager recognizes that the task of defining the characteristics of a widget or component is different from defining how it should appear on the Palette. The Builder Xcessory Object Packager allows you to manipulate the WML data separately from the catalog data. You can load and save files independently, perhaps editing several catalog files in succession while working on one WML file. However, in recognition of the tie between the two files, the Builder Xcessory Object Packager creates a new catalog entry automatically for newly-created widget or component classes and deletes entries for those that are deleted.

Starting the Builder Xcessory Object Packager

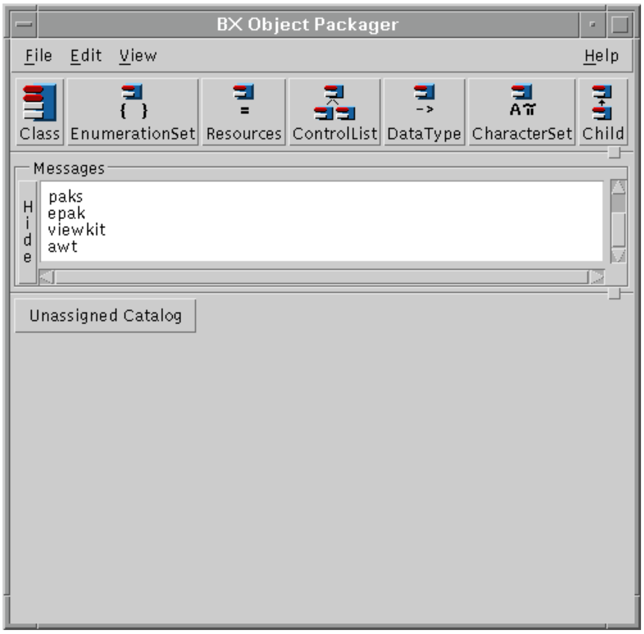
To run the Builder Xcessory Object Packager, enter the following command:

```
% bxop60
```

This script sets the environment variables appropriate for your system and then runs the Builder Xcessory Object Packager binary with the correct `-system_directory` argument. For more information on command-line options, refer to *“Command-line Options and Resources”* on page 62.

Builder Xcessory Object Packager Main Window

Once the BX Object Packager finishes loading, you can access the following BX Object Packager main window (shown here without any data loaded):



paks
epak
viewkit
awt

Figure 1. BX Object Packager Main Window

The following sections describe the components of the interface.

Menubar

The BX Object Packager menubar offers the following menus for controlling the rest of the application.

WML menu	Allows you to open, load, and save WML files, and exit the application.
Edit menu	Allows you to edit WML data.
Catalog menu	Allows you to load and save catalog files, and modify the catalog view.
View menu	Allows you to show/hide the toolbar, show/hide messages, and clear messages.
Help menu	Provides options for obtaining help on the meaning of data and on how to manipulate the application.

Toolbar

Duplicates the Edit menu options with their iconic representations as toolbar entries.

Catalog Editor

Displays catalog hierarchies as follows:

- Catalog hierarchy for classes that have been created but not located into a catalog..
- Editable catalog hierarchy that displays the contents of any loaded catalog file

Message Area

Displays informational messages and warnings for the entire application.

Editing WML Files

Setting data for generation into the WML file involves completing several forms with the data appropriate to each named tag. Almost all the data is string (text) or boolean (on/off) data; there are a few items that are lists of strings.

BX Object Packager Edit Menu

The Edit menu provides several options, each of which is shown on the BX Object Packager Toolbar. Each menu item corresponds to one of the major portions of the WML file. (Refer to *Chapter 8—Modifying the WML File* for more detailed information, or use the Help menu or the F1 key for context-sensitive help information.)

Edit Selector

Selecting a menu item to display the Editor Selector, a list of previously defined values of this type, if any, as shown in the following figure:

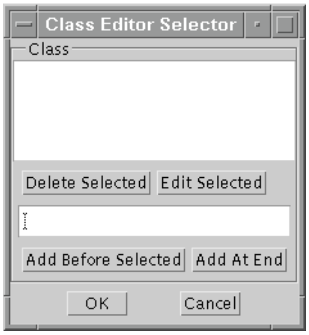


Figure 2. Editor Selector with Two Classes Defined

Adding and selecting an item

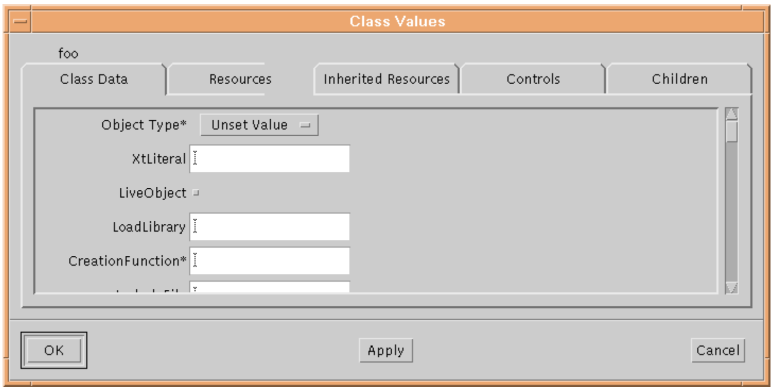
To add a new item, use one of the following methods:

- Select an item by double-clicking on the item, and edit its characteristics.
- Click on the Edit Selected button.

Classes and Resources

The primary screens are for Classes and Resources. Refer to *“Changing Class Information”* on page 64 and *“Changing Resource Information”* on page 86 for more information about using these screens. In general, the most important information is presented first.

Typically, only information marked with an asterisk is required, although your particular objects might demand additional specifications.



Object Type* Unset Value
XtLiteral
LiveObject
LoadLibrary
CreationFunction*

Figure 3. Data Editor Screen

The data screens are designed so that you can back out your changes without modifying the data. The “Cancel” option restores items to their previous settings without affecting the saved data. You can then reselect the item to edit its associated data.

Note: These screens use the Return key as a signal to save the data and to unpost. You should use the Motif traversal mechanisms (the tab key and the arrow keys, or clicking with the mouse) to move keyboard focus within the window.

Loading Data From Widget Libraries

The Class Values editor allows you to define how the Builder Xcessory manipulates the widget or component. You can define resources, automatically-created children, and valid children.

For widgets and gadgets, there are often many resources. If you have the library containing this widget or gadget class in the shared-library form that the Builder Xcessory can load automatically, then you can use the Builder Xcessory Object Packager to load the initial resource definitions directly from the library, rather than enter them all by hand.

Loading a library

To load a library, use the following procedure:

1. Click on the Class button.
2. In the Class Editor Selector, enter the class name for the widget or gadget.
3. Click on Edit Selected to edit the class.
4. On the Class Data screen, check that Object Type is either Widget or Gadget.
5. Set XtLiteral to the name of the widget class (for example, for XmDumbLabel, set it to xmDumbLabelWidgetClass).
6. Select LiveObject, which tells the Builder Xcessory to load the class defined in the WML file automatically from the library when it runs.
7. Specify LoadLibrary to name the library to load it from.

   The Builder Xcessory Object Packager uses the same search path as the Builder Xcessory, so you can probably specify just the name of the library without specifying a full path to it. When you continue to the next item, you will be asked for confirmation about loading the widget or gadget class from the library.

Note: Because Xt modifies widget classes when they are initialized in such a way that the original information in the widget or gadget class cannot be reconstructed, if you load a library that the Builder Xcessory Object Packager itself uses, you will see best-guess data.

---

8. Move to the next field with the Tab key. A dialog appears to confirm that the resources should be loaded.

Background WML Files

The WML portion of the Builder Xcessory Object Packager offers a way to edit a WML file. But the nature of WML files is such that they can exist in an incomplete state. A widget class, for example, can refer to a superclass that is defined in a different WML file (such as `motif.wml`).

While editing a WML file describing widgets or components that you wish to make available to the Builder Xcessory when it runs, you may find that you need to load another WML file to make various datatypes, enumeration sets, and constants available to the Builder Xcessory Object Packager. For this reason, the Builder Xcessory Object Packager offers a mechanism to incorporate those WML files without interrupting the work that you are doing on the WML file that you are editing. These "background WML files" can be incorporated by choosing Merge Background File on the WML menu.

Note: Edit screens that are posted are not updated with the new information. You should apply the changes you have made, dismiss the window, and then re-edit the item in order to see the new data defined in the merged WML file.

Editing the Catalog

You edit the catalog on the main screen (Figure 1 on page 55), setting it up as you want it to appear when Builder Xcessory is run. You can load an existing catalog (and save it independently), or create your own catalog. The interface shows the catalog in the outline form, with menus representing the catalog as a whole and the groups within it. Within a group, items are shown. You can select a menu item from the catalog or group to cut/paste groups or items, or to change the properties of the catalog or group. You can select a popup menu on items to modify them, or move them to the cut buffer. Refer to *Chapter 9—Creating Other Control Files* for information about property values.

The catalog is broken up into two sections. The lower section is a fully-editable catalog. You can create new groups and items, move them about, temporarily save them to the cut-buffer while you load a new catalog, and set the properties on them. It is this section that is saved to the catalog file.

Unassigned Catalog

The Unassigned Catalog displays items that correspond to classes created using the WML editor. It serves as a reminder that these items should be placed in a catalog in order to have them appear when the Builder Xcessory is started and reads in your newly-created files. This catalog offers a more limited set of operations appropriate to its nature as a temporary placeholder for icons.

The catalog file is generated from the lower section, to which you can give your own name. To move catalog entries from the upper "holding pen", select MB3 Cut on the item, then, creating a new group if necessary, paste them into the editable catalog.

The catalog as a whole, and the groups and items that you create, have properties that tie them to classes defined in WML and that affect how Builder Xcessory displays them. For a complete description of these properties, see *“Item Attributes”* on page 111 and *“Groups Attributes”* on page 112.

Select Properties on the catalog, group, or item to change characteristics. You should give the catalog your choice of names, at least.

Command-line Options and Resources

The Builder Xcessory Object Packager allows specification of several data values. These values affect the program's operation and can be specified as command-line options or resources, as shown in the following table:

Command-line Option	Resource	Type	Default
`-filename`	filename	String	none
`-local_directory`	localDirectory	String	as in BX
`-prefix`	prefix	String	XtN
`-system_directory`	systemDirectory	String	as in BX

Resources

The following sections describe the resources:

filename

Specifies the WML file to read at start-up and to operate on.

localDirectory

Specifies the user's local directory, which is used to read files at start-up in addition to the system directory. Usage is as in Builder Xcessory, so the Object Packager has access to the same information and files.

prefix

Specifies the resource name prefix to use for resources loaded dynamically from widget libraries.

systemDirectory

Specifies the system directory containing the Builder Xcessory installation. Usage is the same as in Builder Xcessory, so that the Object Packager has access to the same information and files.

Modifying the WML File

8

Overview

This chapter includes the following sections:

- **WML Files**
- **Changing Class Information**
- **Changing Resource Information**
- **Changing Enumeration Information**
- **Changing DataType Information**
- **Changing Other WML Entries**
- **UIL Data Types**

WML Files

Builder Xcessory uses an extended variant of the OSF/Motif Widget Meta Language (WML) to describe the capabilities of widgets and components. Refer to the *Motif Programmer's Reference, Volume #3* for a thorough overview of Motif WML.

WML file structure

A WML file consists of the following components:

- Keywords that describe the type of data to follow
- Data bounded by braces
- Name-value pairs that describe the data

If you use the Builder Xcessory Object Packager to create or edit the WML file, you are not required to make new WML files for new widgets or class components. However, you might want to hand-edit the file at a later time.

Warning: Use extreme caution when editing WML files. Invalid WML entries may render Builder Xcessory inoperable

Changing Class Information

The class information includes the specific attributes that Builder Xcessory requires to manipulate the widget or component correctly. In addition to the *Motif Programmer's Reference, Volume #3* class directives, Builder Xcessory supports many new directives.

Note: In the following sections, the construction "A | B | C" means "one of the values A, B, or C".

Object Class Diagram

An object class is specified in the WML file as follows:

```
Class <classname> : <classtype> {
   <class attribute>;
   <class attribute>;
   <...>;

   Resources {
      <resource name>;
      <resource name> {
         <resource attribute>;
         <resource attribute><...>;
      };
      <...>;
   };
   Controls {
      <control list name>;
      <object class name>;
      <...>;
   };

   Children {
      <child name> [ Managed | Unmanaged ];
      <...>;
   };
};
```

<classtype> values

In the object class diagram, `<classtype>` can have one of the values shown in the following table:

Value	Description
MetaClass	The object can never be instantiated, and is used only as a base class for other object classes being described in the WML file. A MetaClass is useful for defining a set of attributes common to a number of other object classes.
VkComponent	The object class defines a C++ component based on the ViewKit application framework.
UIComponent	The object class defines a C++ component having the C++ class UIComponent in its superclass hierarchy.
Widget	The object class defines a windowed user interface object based on the Xt Intrinsics.
Gadget	The object class defines a windowless user interface object based on the Xt Intrinsics.

Resources

The Resources section of the class specification lists any resources that the object class adds to those it inherits from its superclasses. You can also use this section to override any of the superclass resource definitions.

Controls The Controls section lists any object classes that this object class accepts as valid children. These can be listed individually or in the form of a ControlList. A ControlList is simply a named list of object class names.

Children The Children section lists all of the child objects that are created by the object and that are manipulable in Builder Xcessory. Later sections of the WML file specify which aspects of the child objects can be modified.

Class Attributes

The following table lists the name-value pairs for the class attributes:

Note: Attributes marked as Not Used are read by Builder Xcessory, but have no effect. These Not Used values are not guaranteed to be written to generated WML files.

Name	Type
Alias	Not used
AlreadyDropsite	Boolean
AlternateParent	Boolean
AttributeFunction	Text
AutoSubclass	Text
Broken	Boolean
ChildDimension	One-of-Many
ChildFetchFunction	Text
ChildFunction	Text
ChildParentFunction	Text
ChildPosition	One-of-Many
ConstraintFunction	Text
ConvenienceFunction	Text
CreatesShell	Boolean

Name (Continued)	**Type**
CreationFunction	Text
DefaultManaged	One-of-Many
DialogClass	Boolean
DocName	Text
GadgetClass	Text
HiddenParent	Text
IncludeFile	Text
InsertOrder	One-of-Many
InterfaceMapFunction	Text
InternalLiteral	Text
InventorComponent	Boolean
LangDir	Text
LinkLibrary	Text
LiveObject	Boolean
LoadLibrary	Text
MaxChildren	Text (Integer)
Movement	One-of-Many
MyDimension	One-of-Many
MyPosition	One-of-Many
NameResource	Text
NeverMenuParent	Boolean
NoTransform	Boolean
ObjectName	Text
ParentClass	Not used
RealClass	Text
RelatedDialogClass	Text
RepaintBkgd	Boolean
ShellType	One-of-Many

Name (Continued)	**Type**
SuperClass	Text
TclAttributeScript	Text
TclCreateScript	Text
TclFile	Text
TclPostCreateScript	Text
UsePositionIndex	Boolean
WidgetClass	Text
WidgetGadgetVariation	Text
WidgetResource	Text
XtLiteral	Text

Class Definitions

The Builder Xcessory-specific class directives have the following definitions:

Note: Builder Xcessory does not use all attributes, and may use some attributes that the OSF WML documentation marks as Not Used. Some values are used by Builder Xcessory to manipulate the widgets or class components, while closely-related values are used by the code generator to generate code that manipulates the widgets or class components.

AlreadyDropsite

Syntax

AlreadyDropsite = True | False;

Used By

Builder Xcessory only.

Description

Indicates that the object class already installs Motif-type Drag and Drop handlers. This is important to note so that Builder Xcessory can properly handle its own drag and drop support.

If unspecified, the value is False.

AlternateParent

Syntax

AlternateParent = True | False;

Used By Unused. Maintained for backward compatibility only.

AttributeFunction

Syntax AttributeFunction = "FunctionName";

Used By Builder Xcessory only.

Description For dynamically loaded objects only. Specifies the function to call when a resource attribute has changed on an instance of this class object.

If unspecified, Builder Xcessory calls XtSetValues on the top widget in the object’s hierarchy.

AutoSubclass

Syntax AutoSubclass = "ComponentClass";

Used By Builder Xcessory only.

Description Indicates that when the user tries to create an instance of this class object, Builder Xcessory should prompt them for the name of a subclass. That is, this class cannot be directly instantiated and must always be subclassed. The ComponentClass string specifies the name of the class to store as the superclass for code generation.

If unspecified, Builder Xcessory allows the user to directly instantiate the object.

Broken

Syntax Broken = True | False;

Used By Builder Xcessory only.

Description Indicates that the widget class in question has problems resetting its resources to default values and that rather than trying to set known default values, Builder Xcessory should simply recreate the object and its children. This is very rarely used or needed.

If unspecified, the value is False.

ChildDimension

Syntax ChildDimension = Required | Optional | Ignored;

Used By Builder Xcessory only.

Description Indicates how child dimension resources (height, width) are treated by this object class and thus how they should be handled by Builder Xcessory.

Required	Builder Xcessory always saves the children’s width and height.
Optional	Builder Xcessory saves children’s width and height only if they have been set.
Ignored	Builder Xcessory never saves the children’s width and height. This object completely controls these resources of its children.

If unspecified, the value is Optional.

ChildFetchFunction

Syntax ChildFetchFunction = "FunctionName";

Used By Code Generator only.

Description Specifies the name of the function to use in generated code to fetch the ID of an automatically created child widget, such as the OK button of a XmMessageBox.

If unspecified, the code generator uses XtNameToWidget to fetch widget IDs.

ChildFunction

Syntax ChildFunction = "FunctionName";

Used By Builder Xcessory only.

Description For dynamically loaded objects only. Specifies the function to call after a child of this object class has been created and before a child of this object class is to be destroyed.

If unspecified, Builder Xcessory does nothing after creating a child or before deleting a child.

ChildParentFunction

Syntax ChildParentFunction = "FunctionName";

Used By Builder Xcessory only.

Description For dynamically loaded objects only. Specifies the function to call to determine the widget ID to use as the parent for any children of this object.

If unspecified, Builder Xcessory uses the top widget in the component’s hierarchy.

ChildPosition

Syntax	ChildPosition = Required | Optional | Ignored;
Used By	Builder Xcessory only.

Description

Indicates how child position resources (x, y) are treated by this object class and thus how they should be handled by Builder Xcessory.

Required	Builder Xcessory always saves the children's x and y coordinates
Optional	Builder Xcessory saves children's coordinates only if they have been set.
Ignored	Builder Xcessory never saves the children's x and y. This object completely controls these resources of its children.

If unspecified, the value is Optional.

ConstraintFunction

Syntax	ConstraintFunction = "FunctionName";
Used By	Builder Xcessory only.

Description

For dynamically loaded objects only. Specifies the function to call to set or get constraint resource values defined by this object class.

If unspecified, Builder Xcessory uses XtSetValues and XtGetValues to set or retrieve values from the object's top widget.

ConvenienceFunction

Syntax	ConvenienceFunction = "FunctionName";
Used By	Code Generator only.

Description

For Widgets and Gadgets only. Indicates the name of the Motif-style creation convenience function for this object class.

If unspecified, the code generator uses XtCreateWidget to create widgets.

CreatesShell

Syntax	CreatesShell = True | False;
Used By	Builder Xcessory only.

Description	Indicates whether or not the object class creates a shell widget when it is instantiated. If True, Builder Xcessory does not attempt to create a shell for the object. If False, Builder Xcessory creates a shell parent for the object if it is to be instantiated without any parent. If unspecified, the value is False.

CreationFunction

Syntax	CreationFunction = "FunctionName";
Used By	Builder Xcessory only.
Description	For dynamically loaded objects only. Specifies the function Builder Xcessory calls to create an instance of the object class. This is a required attribute for dynamically loaded objects.

DefaultManaged

Syntax	DefaultManaged = Never | Always | Managed | Unmanaged;
Used By	Builder Xcessory only.
Description	Specifies how Builder Xcessory needs to handle show and hide requests for instances of this object class.

Never	Builder Xcessory ignores requests to show or hide the object instance. The object is not explicitly shown when instantiated.
Always	Builder Xcessory ignores requests to show or hide the object instance. The object is explicitly shown when instantiated.
Managed	Builder Xcessory allows the user to show or hide the object instance. The object is explicitly shown when instantiated.
Unmanaged	Builder Xcessory allows the user to show or hide the object instance. The object is not explicitly shown when instantiated.

If unspecified, the value is Managed.

DialogClass

Syntax	DialogClass = True | False;
Used By	Builder Xcessory only.
Description	Indicates whether the object class describes a Motif dialog variant, like the XmFormDialog variant of the XmForm widget.

If unspecified, the value is False.

DocName

Syntax	DocName = "string value";
Used By	Unused. Maintained for backward compatibility.

GadgetClass

Syntax	GadgetClass = GadgetClassName;
Used By	Builder Xcessory only.

Description

Specifies the name of the object class to use when the user chooses the Make Gadget option in Builder Xcessory. This attribute only applies to Widget class definitions.

If unspecified, the value of the WidgetGadgetVariation attribute is used. If WidgetGadgetVariation is not specified, the menu choice for Make Gadget is not available.

HideShellParent

Syntax	HideShellParent = ShellClassName;
Used By	Builder Xcessory only.

Description

Indicates that Builder Xcessory should always create a shell of the given class as the parent of this object class instance regardless of what the user requested. Additionally, the Builder Xcessory Browser does not show the shell widget in the instance hierarchy.

If unspecified, Builder Xcessory creates shells as requested by the user and show those shells in the Browser.

IncludeFile

Syntax IncludeFile = "ObjectHeaderFile" | "<ObjectHeaderFile>";

Used By Code Generator only.

Description

Specifies the header file(s) to include in any source code that uses an instance of this object class. If multiple header files are required for the object class, list them in the string separated by spaces.

For example, the VkGraph component requires that three files be included when the component is used. The syntax for the IncludeFile directive for VkGraph is as follows:

IncludeFile = "<Xm/Xm.h> <Sgm/Graph.h> <Vk/VkGraph.h>";

If the include directive should be quoted ("...") rather than bracketed (<...>), specify the filename without the brackets:

```
IncludeFile = "foo.h bar.h"
```

If unspecified, no header files are included for instances of this object class.

InsertOrder

Syntax InsertOrder = RealizedFirst | RealizedLast |
AlwaysFirst | AlwaysLast;

Used By Builder Xcessory and Code Generator.

Description Tells Builder Xcessory how to order the creation of children of this object class.

RealizedFirst	The order in which the composite inserts its children internally is first to last if the composite is realized, but last to first if it is unrealized. Builder Xcessory generates the child list first to last to recreate the proper stacking order in the generated code.
RealizedLast	The order in which the composite inserts its children internally is last to first if the composite is realized, but first to last if it is unrealized. Builder Xcessory generates the child list last to first to recreate the proper stacking order in the generated code.
AlwaysFirst	The order in which the composite inserts its children internally is always first to last, regardless of its realized state.
AlwaysLast	The order in which the composite inserts its children internally is always last to first, regardless of its realized state.

If unspecified, the value is AlwaysFirst.

InterfaceMapFunction

Syntax InterfaceMapFunction = "FunctionName";

Used By Unused. Intended future feature.

InternalLiteral

Syntax InternalLiteral = "SymbolName";

Used By Unused. Maintained for backward compatibility.

InventorComponent

Syntax InventorComponent = True | False;

Used By	Builder Xcessory and Code Generator.
Description	Indicates that the component is a subclass of an OpenInventor object. Builder Xcessory uses this flag to perform special initialization for OpenInventor classes. Likewise, the code generator uses this flag to determine whether any OpenInventor-specific initialization needs to be done in the generated source code. If unspecified, the object class is not treated as an OpenInventor class.

LangDir

Syntax	LangDir = "C" | "CXX" | "VK" | "JAVA" | "C_UIL";
Used By	Builder Xcessory only.
Description	Indicates the code generation languages for which this object can be used. This attribute is maintained principally for backward compatibility. More precise specification of usage is available in the Palette catalog file. If unspecified, the object is assumed to be available for all languages other than Java.

LinkLibrary

Syntax	LinkLibrary = "LibrarySpecification";
Used By	Code Generator only.
Description	Specifies the libraries to include in the generated Makefile and Imakefile for any application that uses an instance of the object class. If unspecified, the code generator does not include any additional library in the Makefile or Imakefile.

LiveObject

Syntax

LiveObject = True | False;

Used By

Builder Xcessory only.

Description

Indicates whether the object class should be dynamically loaded. If True, you must also specify (at a minimum) the CreationFunction and LoadLibrary attributes. If False, the object class must be linked into the Builder Xcessory executable. For VkComponent and UIComponent objects, LiveObject must always be True.

If unspecified, the value is False.

LoadLibrary

Syntax

LoadLibrary = "SharedLibraryName";

Used By

Builder Xcessory only.

Description

Specifies the shared library that Builder Xcessory must load in order to access the functions to create and manipulate this object class.

In most cases, the library can be specified without a full pathname. Builder Xcessory searches a specific set of directories to find the library; however, a fully qualified pathname can be supplied.

Additionally, the library extension does not need to be specified. Currently Builder Xcessory supports the `.so` and `.sl` extensions for shared libraries. If the library specified does not have an extension, Builder Xcessory searches for versions of the library with both the `.so` and `.sl` extensions. For example, the CDE Widget library is libDtWidget.so on most platforms, but libDtWidget.sl under HP-UX. For the CDE widgets, the WML file uses the following LoadLibrary specification:

LoadLibrary = "libDtWidget";

If unspecified, no library is loaded to access the object or any of the functions to manipulate it.

MaxChildren

Syntax	MaxChildren = "NumberOfChildren";
Used By	Builder Xcessory only.
Description	Specifies the maximum number of children that instances of this object class can accept. A value of “0” means that the object class does not accept any children. If unspecified and the object class is a container subclass, the number of children is unlimited.

Movement

Syntax

```
Movement = SetValues | Reorder | UseParent | FormMove |
    Fixed | MenuPaneMove | Configure | MoveWH;
```

Used By Builder Xcessory only.

Description Tells Builder Xcessory how to move children of an instance of this object class.

SetValues	Use XtSetValues to set the X and Y values of the instance.
Reorder	Recreate the entire child hierarchy to reflect a new ordering of the children. This is primarily used in menu container objects.
UseParent	Move this object instance rather than the child object. This is generally used if the object accepts only a single child and completely controls the child’s geometry.
FormMove	Set the XmForm constraint resources (XmNtopOffset, XmNleftOffset, etc.) rather than the geometry resources (XmNx and XmNy) to reposition the child.
MenuPaneMove	Maintain the geometry settings of the child by manipulation of XmPaned window resources. Moving a child widget reorders the child list of this object class and moves the child to the end of the list. This setting is largely unused.
Fixed	Children cannot be moved. This object class completely controls the placement of its child objects.
Configure	Use XtConfigureWidget to move child objects. This setting is largely obsolete and should not be used.
MoveWH	Only allows moving children down (increasing XmNy value) and/or to the right (increasing XmNx value). This option is obsolete and should not be used.

If unspecified, the value is SetValues.

MyDimension

Syntax	MyDimension = Required | Optional | Ignored;
Used By	Builder Xcessory only.
Description	Indicates how dimension resources (height, width) of this object class are to be handled by Builder Xcessory.

Required	Builder Xcessory always saves the object's width and height.
Optional	Builder Xcessory saves objects's width and height only if they have been set.
Ignored	Builder Xcessory never saves the object's width and height.

If unspecified, the value is Optional.

MyPosition

Syntax	MyPosition = Required | Optional | Ignored;
Used By	Builder Xcessory only.
Description	Indicates how position resources (x, y) of this object class are to be handled by Builder Xcessory.

Required	Builder Xcessory always saves the object's x and y coordinates.
Optional	Builder Xcessory saves object's coordinates only if they have been set.
Ignored	Builder Xcessory never saves the object's x and y coordinates.

If unspecified, the value is Optional.

NameResource

Syntax	NameResource = "ResourceName";
Used By	Builder Xcessory only.
Description	Indicates the resource Builder Xcessory should use to display an object's instance name. This is used only if the resource is not set to some other value. In most cases, this option is unnecessary.

If unspecified, Builder Xcessory does not set any resources, but assumes that the object displays its instance name as its default label.

NeverMenuParent

Syntax NeverMenuParent = True | False;

Used By Builder Xcessory only.

Description If True, indicates that, should the user try to create a popup menu child of an instance of this object class, Builder Xcessory should automatically create the menu as a child of this object’s parent. This is used almost exclusively for Gadget classes to avoid setting the necessary event handlers on a windowless object.

If unspecified, the value is False.

NoTransform

Syntax NoTransform = True | False;

Used By Builder Xcessory only.

Description If True, indicates that Builder Xcessory should not allow the user to enter a new object class in the Resource Editor Class field.

If unspecified, the value is False.

ObjectName

Syntax ObjectName = "NameString";

Used By Code Generator only.

Description Used to pass an alternate object class name to the Code Generator. In a code generator Tcl script, the @object_name function is used to retrieve the value of this attribute. It is rarely necessary to use this attribute.

If unspecified, the ObjectName is the same as the Class name.

RealClass

Syntax RealClass = "ObjectClassName";

Used By Builder Xcessory only.

Description Used to indicate that this object class is really just a convenience function that creates an instance of the specified object class.

If unspecified, the object class is assumed to be a true Widget, Gadget, VkComponent, or UIComponent class.

RelatedDialogClass

Syntax	RelatedDialogClass = "ObjectClassName";
Used By	Builder Xcessory only.
Description	Indicates that if this object is created as a child of an XmDialogShell, Builder Xcessory should treat it as an instance of the named object class.

If unspecified, Builder Xcessory assumes that no special handling of an XmDialogShell parent is required.

RepaintBkgnd

Syntax	RepaintBkgnd = True | False;
Used By	Unused. Maintained for backward compatibility.

ShellType

Syntax	ShellType = TopLevel | Dialog | Menu | Meta | ClassShell;
Used By	Builder Xcessory only.
Description	Only used for Shell widgets. Indicates the shell type for this object class.

TopLevel	The object class is an option in the MB1 option menu of Shells panel of the User Preferences dialog.
Dialog	The object class is an option in the MB3 option menu of Shells panel of the User Preferences dialog.
Menu	The object class is a Motif XmMenuShell and should be hidden. This option should almost never be used.
Meta	The object class is a superclass shell that is never to be instantiated. It is principally a placeholder and defines a set of common attributes for shells later in the hierarchy.
ClassShell	This object class is used in Classes view to contain each class being defined in the current project. Never use this setting.

If unspecified, the object is not considered a shell.

SuperClass

Syntax SuperClass = "ClassObjectName";

Used By Builder Xcessory and Code Generator.

Description Indicates the name of the super class of this object class. For example, the XmPushButton is a subclass of XmLabel and inherits the resources specified by the XmLabel. To indicate this, the SuperClass of XmPushButton is set to XmLabel. In the ViewKit class hierarchy, the VkFatalErrorDialog is subclassed from VkErrorDialog. So, its SuperClass value is VkErrorDialog.

If unspecified, the object class is assumed to be the top of a class hierarchy and inherits no resources.

TclAttributeScript

Syntax TclAttributeScript = "TclProcName";

Used By Code Generator only.

Description The name of the code generation Tcl procedure to call in order to process any resources/attributes set on instances of this object class. This is primarily used for non-widget object classes.

If unspecified, the code generator handles resources/attributes using a number of default procedures. For C++ objects, the code generator simply calls XtSetValues on the top-level widget of the object. For Widgets, it calls XtSetValues on the widget instance.

TclCreateScript

Syntax TclCreateScript = "TclProcName";

Used By Code Generator only.

Description The name of the code generator Tcl procedure to call in order to generate the source code necessary to create an instance of this object class. This is primarily used for non-widget object classes.

If unspecified, the code generator uses default methods of creating an instance of this object class. For C++ objects, it calls the operator `new`. For Widget objects, it calls the ConvenienceFunction specified or uses XtCreateWidget if the ConvenienceFunction is not specified.

TclFile

Syntax TclFile = "FileName";

Used By Code Generator only.

Description Specifies the name of a file in `{BX}/xcessory/gen/class` that contains Tcl code generation functions needed when generating source code for this object class.

If unspecified, the functions called when generating source code for an instance of this object class are assumed to be already accessible.

TclPostCreateScript

Syntax TclPostCreateScript = "TclProcName";

Used By Code Generator only.

Description The name of the code generator Tcl procedure to call after the source code creating an instance of this object class has been generated. This allows additional source code to be generated to do any needed work.

If unspecified, no special source code is generated.

UsePositionIndex

Syntax UsePositionIndex = True | False;

Used By Unused. Maintained for backward compatibility.

WidgetClass

Syntax WidgetClass = WidgetClassName;

Used By Builder Xcessory only.

Description Specifies the name of the object class to use when the user chooses the Make Widget option in Builder Xcessory. This attribute only applies to Gadget class definitions.

If unspecified, the value of the WidgetGadgetVariation attribute is used. If WidgetGadgetVariation is not specified, the menu choice for Make Widget is not available.

WidgetGadgetVariation

Syntax	WidgetGadgetVariation = WidgetOrGadgetClassName;
Used By	Builder Xcessory only.
Description	Specifies the name of the object class to use when the user chooses the Make Widget or Make Gadget option in Builder Xcessory. This attribute specifies the Widget class to use if this is a Gadget class and the Gadget class to use if this is a Widget class.

If unspecified, the value of the WidgetClass or GadgetClass attribute is used. If these are not specified, the menu choices for Make Widget and Make Gadget are not available.

WidgetResource

Syntax	WidgetResource = ResourceName;
Used By	Unused. Maintained for backward compatibility.

XtLiteral

Syntax	XtLiteral = "WidgetClassStructureName";
Used By	Builder Xcessory and Code Generator.
Description	Specifies the name of the Xt WidgetClass structure that describes this object class. This option is used only for Widget and Gadget classes. For example, the WidgetClass structure for the Motif PushButton class is xmPushButtonWidgetClass.

If unspecified, Builder Xcessory requires that one of two conditions be true:

- The object class is dynamically loaded and the CreationFunction and LiveObject attributes have been set.
- The object class has been added to Builder Xcessory by recompiling the Builder Xcessory executable and specifying a creation function to use in the call to RegisterWidgetExtended.

Changing Resource Information

To change resource information, change the resource definition in the WML file. While reading the following sections, please refer to the *Motif Programmer's Reference, Volume #3*.

The resource specification tells Builder Xcessory how to manipulate a resource for a widget or component. Resources are globally defined, but specific classes can override various resource attributes. In addition to the *Motif Programmer's Reference, Volume #3* resource directives, many new directives are also supported.

Note: In the discussions of grammar in the following sections, the construction "A | B | C" means "one of the values A, B, or C".

Specifying a resource in the WML file

A resource is specified in the WML file as follows:

```
Resource <resourcename> : <resourcetype> {
   <resource_attribute>
   <resource_attribute>
   <...>
};
```

`<resourcetype>` can have one of the following values:

Argument	Specifies a simple attribute of an object class. Builder Xcessory presents this resource in the Resource Editor list of any instance of any object class that uses the resource.
Constraint	Specifies a constraint attribute of an object class. Builder Xcessory presents this resource in the Resource Editor list of any child of the object class that uses the resource.
Reason	Defines an Xt-style callback.
VkReason	Defines a ViewKit-style callback.

Additionally, the value SubResource is parsed by Builder Xcessory and the other tools, but is unused.

The name-value pairs for the Resource Attributes follow. The same values are also used for the Resources section in a Class definition, where they override, for a specific class, the general attributes of a resource; differences between the two are noted where they exist: (Attributes marked as Not Used are read by Builder Xcessory, but have no effect.)

Note: Values unused by Builder Xcessory are not guaranteed to be written to generated WML files.

Name	**Type**
ActionView	Boolean
Alias	Not used
AllView	Boolean
AllowEmptyValue	Boolean
AlwaysDefault	Boolean
AlwaysOutput	Boolean
AlwaysSetValues	Boolean
AppDefaults	Boolean
AugmentDefault	Text
AutoSet	Boolean
BeView	Boolean
CallbackFunc	Boolean
CodeOutput	Boolean
CreationSet	Boolean
CustomView	Boolean
DbResource	Boolean
Default	Text
DocName	Text
DropFunc	Boolean
EnumerationSet	Text
Exclude	Boolean
Excuse	Text
Expose	Boolean
Forced	Boolean
FreeConvert	Text

Name (Continued)	**Type**
FuncDef	Text
FuncProto	Text
GetRoutine	Text
Ignore	Boolean
Insensitive	Boolean
InternalLiteral	Text
KeepAlloc	Boolean
KeepOnMove	Boolean
LangDir	Text
LastWordConvert	Boolean
MethodName	Text
NeverSet	Boolean
NeverVerify	Boolean
OverrideDefault	Text
ReadInitialValue	Boolean
ReadOnly	Boolean
Recreate	Boolean
RecreateParent	Boolean
Related	Text
RelatedFont	Text
ResourceLiteral	Text
SetRoutine	Text
TclScript	Text
ThrowAwayOnPaste	Boolean
Type	Text
TypeName	Text
TypeSize	One-of-Many
UnderscoreConvert	Boolean

Name (Continued)	Type
UpdateAllRsc	Boolean
VisualView	Boolean
WlShellsOnly	Boolean
WlSkipSelf	Boolean
WlUseAll	Boolean
WlUseClasses	List
WlUseSOSC	Boolean
WlUseSiblings	Boolean
XrmResource	Boolean

The following sections define the Builder Xcessory resource directives listed in the previous table. Builder Xcessory does not use all attributes. In particular, some attributes are retained by Builder Xcessory only for backward compatibility. Also, Builder Xcessory may use some attributes marked Not Used in the OSF WML documentation.

Some values are used by Builder Xcessory to control how Builder Xcessory manipulates or displays the resource, while closely-related values are used by the code generator to emit code that manipulates the widgets or class components.

ActionView

Syntax AllView = True | False;

Used By Builder Xcessory only.

Description Specifies that the resource is displayed in Resource Editor when the Actions/Callbacks option is set.

If unspecified, the value is False. This item should be set to True if the resource is a callback.

AllView

Syntax AllView = True | False;

Used By Builder Xcessory only.

Description Specifies that the resource is displayed in Resource Editor when the view option is set to All Resources.

If unspecified, the value is True.

AllowEmptyValue

Syntax AllowEmptyValue = True | False;

Used By Builder Xcessory only.

Description Specifies that a “NULL” string is a valid value for the resource and should not be interpreted as resetting the value to its default value.

If unspecified, the value is False.

AlwaysDefault

Syntax AlwaysDefault = True | False;

Used By Unused. Maintained for backward compatibility.

AlwaysOutput

Syntax AlwaysOutput = True | False;

Used By Builder Xcessory only.

Description Specifies that Builder Xcessory should always save the value of the resource to the save file.

If unspecified, the value is False.

AlwaysSetValues

Syntax AlwaysSetValues = True | False;

Used By Builder Xcessory and Code Generator.

Description When setting the resource value internally and in generated code, Builder Xcessory never passes the value to the creation routine. Rather, the resource value is set after the object has been created.

If unspecified, the value is False.

AppDefaults

Syntax AppDefaults = True | False;

Used By Builder Xcessory only.

Description Specifies that the resource can be specified in an application defaults file.

If unspecified, the value is True.

AugmentDefault

Syntax	AugmentDefault = "DefaultValue";
Used By	Unused. Maintained for backward compatibility.

AutoSet

Syntax	AutoSet = True | False;
Used By	Unused. Maintained for backward compatibility.

BeView

Syntax	BeView = True | False;
Used By	Builder Xcessory only.
Description	Specifies that the resource modifies the widgets behavior.When set to True, the resource will be displayed in the resource editor when the view is set to "Behavior View" If unspecified, the value is False.

CallbackFunc

Syntax	CallbackFunc = "FunctionName";
Used By	Unused. Maintained for backward compatibility.

CodeOutput

Syntax	CodeOutput = True | False;
Used By	Builder Xcessory only.
Description	Specifies that the resource can be hard-coded in the generated source code. If unspecified, the value is True.

CreationSet

Syntax	CreationSet = True | False
Used By	Code Generator only.
Description	Used only for Constraint resources. Specifies whether the resource should be set

when an object is created.

If unspecified, the value is False.

CustomView

Syntax	CustomView = True | False;
Used By	Builder Xcessory only.
Description	A misnomer. Specifies that the resource is displayed in Resource Editor when the view option is set to Modified Resources.

If unspecified, the value is True.

Default

Syntax	Default = "ValueString";
Used By	Builder Xcessory only.
Description	Indicates the default value of the resource.

DocName

Syntax	DropFunc = "String";
Used By	Unused. Maintained for backward compatibility.

DropFunc

Syntax	DropFunc = "FunctionName";
Used By	Unused. Maintained for backward compatibility.

EnumerationSet

Syntax	EnumerationSet = EnumerationSetName;
Used By	Builder Xcessory only.
Description	Defines the EnumerationSet to use as the set of values for this resource. For example, the XmForm constraint resource XmNbottomAttachment uses the enumeration set name “Attachment” as the set of all possible values.

If unspecified, Builder Xcessory uses a text field for the user to enter the resource value, not a One of Many editor.

Exclude

Syntax	Exclude = True | False;
Used By	Builder Xcessory only.
Description	Indicates that Builder Xcessory should hide this resource. This attribute is used only in a Class specification’s Resources section. For example, the XmNautoUnmanage resource that all subclasses of XmBulletinBoard use is only used when the widget is a dialog variant (XmFormDialog versus XmForm). In the Resources section of XmForm, the XmNautoUnmanage resource is set to Exclude = True to hide the resource. If unspecified, the value is False.

Excuse

Syntax	Excuse = "String";
Used By	Builder Xcessory only.
Description	Specifies a string to print instead of allowing a resource value to be set. For example, the XmNchildren resource is not allowed to be set, so Excuse is set to Read Only. If unspecified, the resource value can be set.

Expose

Syntax	Expose = True | False;
Used By	Builder Xcessory only.
Description	Specifies whether or not the resource can be exposed when creating class components in Builder Xcessory. If unspecified, the value is True.

Forced

Syntax	Forced = True | False;
Used By	Unused. Maintained for backward compatibility.

FreeConvert

Syntax	FreeConvert = "MemoryFreeFunctionName";
Used By	Code Generator only.

Description	Specifies the function to call in the generated code in order to free values for this resource that are converted from string values using an Xt-style converter.

Functions that can be used in this way are void functions taking a single argument of a pointer to the type to be freed.

If unspecified, converted values are not freed in generated code. In most cases, this is the desired behavior. However, some widgets copy certain resource values (such as the XmLabel with values of the XmNlabelString resource), so that memory allocated by the converter needs to be freed.

FuncDef

Syntax	FuncDef = "String";
Used By	Unused. Maintained for backward compatibility.

FuncProto

Syntax	FuncProto = "String";
Used By	Unused. Maintained for backward compatibility.

GetRoutine

Syntax	GetRoutine = "String";
Used By	Unused. Maintained for backward compatibility.

Ignore

Syntax	Ignore = True | False;
Used By	Builder Xcessory only.
Description	Specifies that a given resource should not appear in the resource list of instances of a particular class.

If unspecified, the resource is listed where appropriate.

Insensitive

Syntax	Insensitive = True | False;
Used By	Unused. Maintained for backward compatibility.

InternalLiteral

Syntax	InternalLiteral = "SymbolName";
Used By	Unused. Maintained for backward compatibility.

KeepAlloc

Syntax	KeepAlloc = True | False;
Used By	Builder Xcessory only.
Description	Tells Builder Xcessory not to free memory allocated when this resource is set internally. If unspecified, Builder Xcessory frees the internally allocated memory.

KeepOnMove

Syntax	KeepOnMove = True | False;
Used By	Unused. Maintained for backward compatibility.

LangDir

Syntax	LangDir = "C" | "CXX" | "VK" | "JAVA" | "C_UIL";
Used By	Builder Xcessory only.
Description	Indicates the code generation language for which this resource can be used. If unspecified, the resource is assumed to be available for all languages other than Java.

LastWordConvert

Syntax	LastWordConvert = True | False;
Used By	Builder Xcessory only.
Description	Use the last “word” of the enumeration symbol name of the current value of this resource as the string value in an application defaults file. The last word begins at the last capitalized letter in the symbol name and continues to the end of the symbol name. This is very rarely used. If unspecified, the value is False.

MethodName

Syntax MethodName = "MethodNameString";

Used By Code Generator only.

Description The name of the class method to use to set the value of this resource on an instance of a class object. The set method has no return value and takes one argument—the value to set.

If MethodName is unspecified, the code generator uses the TclAttributeScript of the object class to set values. If neither of those is set, the code generator uses XtSetValues on the widget or the top-level widget of a C++ component.

NeverSet

Syntax NeverSet = True | False;

Used By Builder Xcessory only.

Description Tells Builder Xcessory to never set the resource value.

If unspecified, Builder Xcessory always tries to set the value on the object instance in question.

NeverVerify

Syntax NeverVerify = True | False;

Used By Builder Xcessory only.

Description Indicates that Builder Xcessory should accept the value entered by the user as the value for the resource and not attempt to fetch the resource value from the object instance after setting it.

OverrideDefault

Syntax OverrideDefault = "DefaultValue";

Used By Builder Xcessory only.

Description Indicates that Builder Xcessory should not allow the user to set this resource value and to display the given DefaultValue as the resource value. This is only set on resource specifications in the Resources section of a Class specification.

ReadInitialValue

Syntax ReadInitialValue = True | False;

Used By	Unused. Maintained for backward compatibility.

ReadOnly

Syntax	ReadOnly = True | False;
Used By	Builder Xcessory only.
Description	Indicates that the resource cannot be set by the user. Its value is displayed by Builder Xcessory.

Recreate

Syntax	Recreate = True | False;
Used By	Builder Xcessory only.
Description	Indicates that Builder Xcessory must recreate the object instance whenever it sets this resource value. Essentially, this is used to indicate creation-time only resources.

RecreateParent

Syntax	RecreateParent = True | False;
Used By	Builder Xcessory only.
Description	Indicates that Builder Xcessory must recreate the object instance’s parent whenever it sets this resource value.

Related

Syntax	Related = ResourceName;
Used By	Builder Xcessory and Code Generator.
Description	Names a resource for which this resource acts as a counter.

RelatedFont

Syntax	RelatedFont = ResourceName;
Used By	Builder Xcessory only.
Description	Names a XmFontList resource that provides the font tags used by this XmString resource.

ResourceLiteral

Syntax	ResourceLiteral = "String";
Used By	Unused. Maintained for backward compatibility.

SetRoutine

Syntax	SetRoutine = "SetEnabled" | "SetListResource" | "SetFormAttachment" | "SetFormOffsetPosition" | "SetRecomputeSize" | "SetSensitive" | "SetTearOffModel";
Used By	Builder Xcessory only.
Description	Tells Builder Xcessory to use one of several special routines to set this resource. Avoid using this attribute.

TclScript

Syntax	TclScript = "TclProcName";
Used By	Code Generator only.
Description	Specifies a tcl function that the code generator should call when generating source code to set this resource value.

ThrowAwayOnPaste

Syntax	ThrowAwayOnPaste = True | False;
Used By	Builder Xcessory only.
Description	Indicates that Builder Xcessory should not try to set this resource value when pasting an instance of an object class that originally set this resource value.

Type

Syntax	Type = WMLDataType;
Used By	Builder Xcessory and Code Generator.
Description	Matches the resource to a defined data type. Builder Xcessory uses this attribute to determine default attributes for the resource as well as to match the resource to an editor. This attribute is required.

TypeName

Syntax TypeName = "ConverterTypeName";

Used By Builder Xcessory and Code Generator.

Description This value is the `to_type` used by XtConvertAndStore. Builder Xcessory and the code generator use this when converting strings to actual resource values. For example, the basic widget resource XmNheight has a TypeName of "Dimension".

TypeSize

Syntax TypeSize = sizeofBoolean | sizeofChar | sizeofDouble | sizeofFloat | sizeofInt | sizeofLong | sizeofPointer | sizeofShort;

Used By Builder Xcessory only.

Description Indicates the size in bytes of the variable that holds the value of this resource. The actual size of the type is determined dynamically by Builder Xcessory, and can vary from platform to platform. It is important that this size be correct for each resource so that Builder Xcessory can allocate sufficient space to hold a resource value and also so that values can be correctly interpreted.

UnderScoreConvert

Syntax UnderScoreConvert = True | False;

Used By Builder Xcessory only.

Description Use the last “word” preceded by an underscore character (_) of the enumeration symbol name of the current value of this resource as the string value in an application defaults file. This is very rarely used.

If unspecified, the value is False.

UpdateAllRsc

Syntax UpdateAllRsc = True | False;

Used By Builder Xcessory only.

Description If this resource value is changed, forces the Resource Editor to update all the displayed data.

VisualView

Syntax VisualView = True | False;

Used By	Builder Xcessory only.
Description	Used by Builder Xcessory to indicate that the resource affects the visual appearance of the widget. This value should be set to True for any visually related resource. The widget will be displayed in the resource editor when the "View - Visual Resources" toggle is set. If unspecified, the value is False

WlShellsOnly

Syntax	WlShellsOnly = True | False;
Used By	Builder Xcessory only.
Description	The extended editor for this resource presents a list of all of the shells currently instantiated by Builder Xcessory.

WlSkipSelf

Syntax	WlSkipSelf = True | False;
Used By	Builder Xcessory only.
Description	The extended editor for this resource never lists the object instance for which this resource is being set.

WlUseAll

Syntax	WlUseAll = True | False;
Used By	Builder Xcessory only.
Description	The extended editor for this resource presents a list of all of the object instances currently instantiated by Builder Xcessory.

WlUseClasses

Syntax	WlUseClasses = ClassName [| ClassName...];
Used By	Builder Xcessory only.
Description	The extended editor for this resource presents a list of all of the object instances of the given object classes currently instantiated by Builder Xcessory.

WlUseSOSC

Syntax	WlUseSOSC = True | False;
Used By	Builder Xcessory only.
Description	The extended editor for this resource presents a list of all of the grandchildren of the object instance for which this resource is being set.

WlUseSiblings

Syntax	WlUseSiblings = True | False;
Used By	Builder Xcessory only.
Description	The extended editor for this resource presents a list of all of the siblings of the object instance for which this resource is being set.

XrmResource

Syntax	XrmResource = True | False;
Used By	Unused. Maintained for backward compatibility.

Changing Enumeration Information

Enumeration set definitions register the named constants used in Builder Xcessory to specify some resource values. Typically these are integer values in some finite range. For example, under the EnumerationSet section of the WML file you might make the following definition.

```
!
! UIL Enumeration Sets
!
EnumerationSet
   ArrowDirection : integer
   {
      XmARROW_UP;
      XmARROW_DOWN;
      XmARROW_LEFT;
      XmARROW_RIGHT;
   };
```

where `XmARROW_UP`, `XmARROW_DOWN`, `XmARROW_LEFT`, and `XmARROW_RIGHT` correspond to enums or #defines in C code.

In the resource section of the WML file, you can define a resource:

```
Resource
   XmNarrowDirection : Argument
   {
      Type = integer;
      EnumerationSet = ArrowDirection;
   };
```

For more information see the “Enumeration Set Definitions” in the *OSF/Motif Programmer’s Reference*.

Builder Xcessory supports the following syntax:

```
<identifier>:”<double quoted string>” = <integer>
```

For example:

```
EnumerationSet
ArrowDirection : integer
{
   XmARROW_UP:”ARROW_UP”=0;
   XmARROW_DOWN:”ARROW_DOWN”=1;
   XmARROW_LEFT:”ARROW_LEFT”=2;
   XmARROW_RIGHT:”ARROW_RIGHT”=3;
};
```

If you omit the =<integer> from an EnumerationSet entry, Builder Xcessory assumes its value to be one greater than the preceding value. The initial value is assumed to be zero.

Changing DataType Information

If any objects you add to Builder Xcessory use a new data type, you need to specify a WML DataType. A DataType is specified in the WML file as follows:

```
DataType <datatype name> {
  <datatype attribute>;
  <datatype attribute>;
  <...>;
};
```

DataTypes can have the following attributes:

CanBeApp

Syntax	CanBeApp = True | False;
Used By	Builder Xcessory only.
Description	Resource values of this type can be put in an application defaults file.

CanBeCode

Syntax	CanBeCode = True | False;
Used By	Builder Xcessory only.
Description	Resource values of this type can be hard-coded in the generated source code.

CanBeEmpty

Syntax	CanBeEmpty = True | False;
Used By	Builder Xcessory only.
Description	The empty (NULL) string is acceptable for resource values of this type. Builder Xcessory should not interpret a NULL value as meaning to reset the resource value to its default value.

TypeName

Syntax	TypeName = "ConverterTypeName";
Used By	Builder Xcessory and Code Generator.
Description	This value is the `to_type` used by XtConvertAndStore. Builder Xcessory and the code generator use this when converting strings to actual resource values. For example, the basic datatype, integer, has a TypeName of "Int".

TypeSize

Syntax	TypeSize = sizeofBoolean | sizeofChar | sizeofDouble | sizeofFloat | sizeofInt | sizeofLong | sizeofPointer | sizeofShort;
Used By	Builder Xcessory only.
Description	Indicates the size in bytes of the variable that holds the value of this datatype. The actual size of the type is determined dynamically by Builder Xcessory, and can vary from platform to platform.

In addition to the aforementioned attributes, all other attributes defined in the OSF/Motif documentation for DataTypes are parsed, but ignored by Builder Xcessory.

Changing Other WML Entries

Typically, the WML file also contains sections for CharacterSet, Children, and Control List. Builder Xcessory uses these entries as described in the OSF/Motif documentation.

UIL Data Types

This section contains additional information about the WML file format that will be useful should hand-editing be necessary. Builder Xcessory uses the following UIL data types:

Note: Values are entered as strings and converted to the correct format with the XtConvert mechanism.

UIL Data Type	Description
`asciz_table`	Multibyte character string table char **.
`boolean`	1 Byte Xt Boolean. For 4 byte Booleans, use integer.
`color`	Any pixel value.
`compound_string`	Motif XmStrings only.
`float`	Double float.
`font`	Any font-like structure, e.g., XFontStruct.
`font_table`	Motif XmFontList.
`identifier`	Default value for unknown types.
`integer`	Basic integer value, either 1, 2, 4, or 8 bytes.
`keysym`	Motif Keysym only.
`pixmap`	Pixmap of screen depth.
`reason`	Any callbacks.
`single_float`	Single float.
`string`	Equivalent to char *.
`string_table`	Table of XmStrings only.
`translation_table`	Translation or accelerator table.
`wchar`	Wide character string.
`widget_ref`	Any widget reference.

Creating Other Control Files

9

Overview

This chapter includes the following sections:

- **Builder Xcessory Control Files**
- **Catalog File**
- **Collection File**
- **Control File**
- **Pixmap File**

Builder Xcessory Control Files

Builder Xcessory uses the WML file to specify how to manipulate the widget classes and user-interface components that you have added. However, Builder Xcessory uses other control files as well. These files specify which WML files should be read by Builder Xcessory upon startup, under what conditions, and how the objects should appear on the Palette. The following table describes the control files:

Control File	Description
Catalog	Identifies the conditions under which the items named in the WML files appear on the Palette.
Collection	Identifies the hierarchy and extra attributes that Builder Xcessory should use to create an instance of the item; specified in the catalog file.
Pixmap	Identifies the icon image that should appear when an item in the WML file appears on the Palette, as specified in the catalog file.
Tcl	Identifies which WML files to include.
WML	Describes how to manipulate the widget classes and user-interface components.

File extensions When files refer to one another, they omit the extensions, for example "`.tcl`" or "`.wml`" or "`.pix`". Builder Xcessory uses the extension appropriate to the file type.

Installed files When a file is described as installed into `{BX}/some_directory`, the file might be installed into that directory, as a system file. Or the file might be installed into your own area in the local `.builderXcessory6` directory, with a similar directory structure.

Catalog File

The Catalog file is a textual description of how the Builder Xcessory Palette should appear. The Palette shows multiple groups of items. For example, all Motif Container widget classes are shown together.[1]

Contents of catalog file

The Catalog file contains information on the name given to the Catalog, including the following information:

- List of groups
- Name that should be shown with each group
- Interface items that initially appear within the group

If you use the Builder Xcessory Object Packager to create a catalog file, you will see how to locate items by direct manipulation within groups and how to set their attributes. You may modify the catalog file later, or create the file manually.

Catalog File Format

Catalog files use the following format:

```
<catalog attribute>;
<catalog attribute>;
<...>;

Group <groupname> [ : <condition> ] {
          <group attribute>;
          <group attribute>;
          <...>;

          Items {
             <itemname> [ : <condition> ] {
                <item attribute>;
                <item attribute>;
                <...>;
             };
             ! ... additional groups or items ...
          };
          ! ... additional groups or items ...
};
!... additional groups ...
```

The file begins with attributes that apply to the catalog as a whole. A list of Palette groups and the contents of those groups follows the catalog attributes.

1. In the default view.

Group specifications

A Group specification lists all of the items and other groups that are to be displayed as part of a Palette group of the given name. Each group can include an optional condition that is used by Builder Xcessory to determine whether to show a group and its contents.

Item specifications

Groups can contain Items and Group specifications. The Items section lists the actual Palette items that correspond to an object class in a WML file. Similar to a Group specification, items can include an optional condition that is used by Builder Xcessory to determine whether to show the item.

We'll use the first part of the file for the Motif widgets, provided by Builder Xcessory, as an example:

```
!
! Default Motif Widget Catalog
!
DocName = "Motif Widgets";

Group motif_containers : Language != "JAVA"  {
          DocName = "Motif Containers";
          DefaultState = Open;
          Items {
             xm_main_window {
               Class = "XmMainWindow";
               DocName = "XmMainWindow";
               PixmapFile = "MainW";
               LargePixmapFile = "MainW_large";
               CollectionFile = "MainW";
               Exported;
             };
              xm_scrolled_window {
                Class = "XmScrolledwindow";
                DocName = "XmScrolledwindow";
                PixmapFile = "ScrolledW";
                LargePixmapFile = "ScrolledW_large";
                CollectionFile = "ScrolledW";
                Exported;
             };
              xm_paned_window {
                Class = "XmPanedWindow";
                DocName = "XmPanedWindow";
                PixmapFile = "PanedW";
                LargePixmapFile = "PanedW_large";
                CollectionFile = "PanedW";
                Exported;
              };

             ! other items not shown

          };
};
```

Item Attributes

You can use the following attributes in an item specification:

Class

Syntax Class = "WMLClassName";

Description Specifies the object class as named in a WML file. This attribute is required for every item.

CollectionFile

Syntax CollectionFile = "FileBaseName";

Description Specifies the collection file to use when creating an instance of this object. Builder Xcessory appends ".col" to the given FileBaseName and searches:

```
${HOME}/.builderXcessory6/collections
```

followed by

```
{BX}/xcessory/collections
```

when looking for the file.

The collection file is a simple UIL file that specifies resources to set on each instance of the object class that Builder Xcessory creates. The Object Packager generates a collection file for every object class that you add.

DocName

Syntax DocName = "String";

Description Specifies the string to display in the label that the Palette pops up over the item when the Palette is in Pixmaps Only view.

Exported

Syntax Exported;

Description Currently unused by Builder Xcessory. Exported indicates visibility to other users in the future.

LargePixmapFile

Syntax LargePixmapFile = "FileBaseName";

Description Specifies a pixmap file basename to use as the icon for the object on the Palette when -largeIcon was passed to BX at execution. This specifies a pixmap with a size of

32x32, which is similar to the PixmapFile in all regards but size. This is a required field. It has been created to help those who may be unable to see smaller pixmaps.

PixmapFile

Syntax PixmapFile = "FileBaseName";

Description Specifies the pixmap file basename to use as the icon for the object on the Palette. Pixmap files have the `.pix` extension and can be located in either `${HOME}/.builderXcessory6/pixmaps` or `{BX}/xcessory/pixmaps`. Builder Xcessory uses a default pixmap for any items that do not specify a PixmapFile attribute or for which the PixmapFile cannot be found.

Pixmap files are XPM Version 3 format files. To be consistent with the other pixmaps on the Palette, icons should be 20x20 pixels and use no more than 8 colors. All of the pixmaps provided with Builder Xcessory use no more than 7 common colors and a separate color as to act as a common group background. You can use `{BX}/xcessory/pixmaps/BXTemplate.pix` as a starting point for your own icons.

Groups Attributes

Groups specifications can use the following attributes.

DefaultState

Syntax DefaultState = Open | Closed;

Description Specifies whether the group should be fully displayed (Open) or displayed only as its name or DocName (Closed).

DocName

Syntax DocName = "String";

Description Specifies the string to display in the Palette to identify the group. When the Palette uses the Outline View, this string is displayed next to the folder button. In the Tabbed View, the string is displayed on the tab corresponding to this group.

Catalog Attributes

The following attributes apply to catalog files:

DocName

Syntax DocName = "String";

Description Specifies the name to use for the catalog as whole. This string is displayed in the Palette window’s title bar if the catalog is loaded into an empty Palette.

Include

Syntax Include "CatalogFileName";

Description Specifies another catalog file that should be included whenever this catalog file is loaded. The CatalogFileName can be either a fully qualified pathname (such as `/usr/local/ui/extra.cat`) or simply the name of the catalog (such as extra.cat). In the former case, Builder Xcessory simply loads the specified file. In the latter case, Builder Xcessory looks in both `${HOME}/.builderXcessory6/package` and `{BX}/xcessory/package` for a file of the given name.

Conditions

Both the item and the group can optionally name conditions under which they should appear on the Palette. Conditions consist of one or more tests combined using logical AND (&&) and inclusive OR (||) operators. You can specify groupings and precedence using parentheses. If no groupings are specified, tests are performed from left to right.

Test types The valid tests are of two types:

- Unary tests for the truth of a state
- Binary tests of two values

Unary tests Unary tests are specified using SystemAttribute(<tag>). If the attribute to which the <tag> refers is set as true, SystemAttribute resolves as true. You can negate the test using the ’!’ character.

Valid System Attribute tags

The following table lists and defines valid SystemAttribute tags:

Table 1:

Tag	Definition
DXm	True if user specified -dec, or loadDEC resource is True.
EPak	True if user specified -ics, or loadICS resource is True.
IrisGL	True if user specified +openGL, or useOpenGL resource is False.
OpenGL	True if user specified -openGL, or useOpenGL resource is True.

Binary tests

Binary tests compare a tag attribute to a value. You can make comparisons using the following operators:

- == for equality
- != for inequality
- < for less than
- > for greater than
- >= for greater than or equal
- <= for less than or equal

Note: You can only make the greater than/less than comparisons on numeric attributes.

Binary valid tags

The following table lists valid tags in comparisons:

Table 2:

Tag	Definition	Operators
DatabaseName	Test the value of the databaseName resource.	==, !=
Env(var)	Test the value of the shell environment variable named "var".	==, !=
Language	Test the value of the user’s selected code generation language.	==, !=

Table 2:

Tag	**Definition**	**Operators**
Platform	Test the name of the operating system on which BX is running. This is the name returned by the uname() system call.	==, !=
Version	Test the version number of the operating system on which BX is running. This will be the version returned by uname().	==, !=, <, >, >=, <=

Collection File

The Collection file is a UIL file that describes the hierarchy used to create the object space.The following example illustrates a sample collection file:

```
module main_uil
names = case_sensitive

object new_item : NewItem {
                arguments { };
                controls { };
                callbacks { };
};
end module;
```

Replace "`new_item`" with the item name from the Catalog file, and "`NewItem`" with the class name from the WML file. This file should be placed in the directory `{BX}/xcessory/collections` or `${HOME}/.builderXcessory6/collections` with the name `<ClassName>.col`.

Control File

This file resides in the directory `{BX}/xcessory/tcl` or `${HOME}/.builderXcessory6/tcl` and is used to load the WML definition file for your components into Builder Xcessory. The following example illustrates a sample control file:

```
@add_wml_file new_items
```

The name of this file must end in the string "`_ctrl.tcl`". At start-up Builder Xcessory reads all "`_ctrl.tcl`" files in the `{BX}/xcessory/tcl` and `${HOME}/.builderXcessory6/tcl` directories to discover which WML files it needs to load. The WML file referenced by this example is either `{BX}/xcessory/wml/new_items.wml` or `${HOME}/.builderXcessory6/wml/new_items.wml`.

Pixmap File

Note: This is an optional file.

The files must be in XPM format version 3 and should be 32x32 pixels in size. If you are working on 8-bit video displays, use as few colors as possible. All pixmaps shipped with Builder Xcessory 6.0 have been redesigned to use a common set of 7 colors and an additional background color to distinguish object sets. A basic pixmap template defining these colors can be found in `{BX}/xcessory/pixmaps/BXTemplate.pix`. The files should be in the directory `{BX}/xcessory/pixmaps/`.

Using Custom Objects 10

Overview

Note: This chapter provides a feature checklist of object functionality. Read this chapter if you intend to incorporate your own objects into Builder Xcessory.

This chapter includes the following sections:

- **Primitive and Manager Classes**
- **Composite Widget Classes**
- **Resources**
- **Objects that Control Specific Children**

Primitive and Manager Classes

Verifying an object class

To verify that an object class works correctly, confirm the following conditions:

- The object class can be instantiated in Builder Xcessory.

Create an instance of your object. It is advisable to try creating your object as a child of different parents, preferably all those that are allowed.

- The class name appears correctly in the Resource Editor.

Verify this manually.

- The code is output correctly, has the correct include file for C/C++ output, compiles, and runs.

Composite Widget Classes

Verifying children in controls list

To verify that the correct children are in the controls list, perform the following operations:

- The default controls list specifies all objects that are possible children (AllWidgets). If your object can have gadget children, change this to include gadgets. (AllWidgetsAndGadgets).
- If your object can have only specific children, edit the controls list by hand.
- Check that the number of children is correctly set. By default any object that can have children can have an unlimited number of children (see *“Objects that Control Specific Children”* on page 122).

Resources

Ensuring that users can modify objects added to Palette

To ensure that users can modify all resources of objects you add to the Palette, verify the following conditions:

- Each resource can be modified.

  Change each resource in the Resource Editor. The object should redisplay itself correctly. Certain resources are defined as “Creation Only”. If this is the case, try setting the Recreate directive in the Resource definition section of your WML file, as follows:

  ```
  Recreate = True;
  ```
- The (...) editor works for each resource.
- Each resource is output correctly to C or C++, and UIL and compiles and runs correctly.
- Each resource is output correctly to `app-defaults`.
- The object runs correctly. Also, some resources are not specifiable in the `app-defaults` file. If this is the case, modify the WML file accordingly, using the AppDefault directive.
- Each resource can be saved and read in correctly.
- If you use UIL as more than a save format, rebuild your UIL compiler.
- If Builder Xcessory cannot determine the correct data type, it defaults to identifier. You may need to change this to a type appropriate for your widget (that is, boolean, integer, string_table).

Objects that Control Specific Children

By default, objects that can have children are defined to accept children of any type. This may not be appropriate for your object. It is possible to redefine this behavior by modifying the Controls section of the object class definition. For example, the section:

```
Controls
{ AllWidgets; };
```

becomes:

```
Controls
{ WidgetClass1; WidgetClass2; WidgetClass3; };
```

where WidgetClassN are the class names of the possible children.

Index

Index

Index

Index

Index

Index