Builder Programmer's Reference Guide

This manual was produced using Applixware.

Printed: February 2000

Contents

Chapter 2 Object-Oriented Concepts

Chapter 3 Editing Object Methods

Chapter 12 Using TableClass Methods

Chapter 13 Using ButtonClass Methods

Chapter 14 Using EntryFieldClass Methods

Chapter 23 Using SpreadsheetsClass, WordsClass, and Graphics Class Methods

Chapter 24 Building Sample Applications

Appendix A Sequence of Events

Figures

Tables

Preface

About This Manual

This manual covers the following topics:

- Language elements used in Applix Builder applications
- Object-oriented concepts and how they apply to Applix Builder applications
- Building simple and advanced applications
- Using the significant methods available in the Applix Builder classes

Audience

The projected audience for the Applix Builder manuals and on-line help are application developers interested in creating graphical, data-centric applications. Applix Builder is a developer's environment for creating applications that are capable of running independent of Applix Builder.

The manual assumes the following:

- Entry level programming experience.
- Basic knowledge of object-oriented programming concepts.
- Basic knowledge of Applix ELF (Extension Language Facility).
- Basic knowledge of SQL.

Organization of This Manual

This manual is organized as follows:

Subject	Chapters
Language elements	1
Object-oriented concepts and using them in applications	2, 3
Application examples	4, 24
Applix Builder classes and significant methods	5-23

Refer to the *Applix Builder User's Guide* for information about using the Applix Builder tools and features.

Conventions Used in This Manual

The following typeface conventions are used throughout this manual:

Helvetica	Helvetica text indicates that this option or object appears in the document window. For example, "Type the name of the document in the File name entry area." File names and directories are also indicated by Helvetica text. For example, "This file is located in your axhome directory." Applixware keys are printed in a Helvetica uppercase typeface. For example, "Press the TAB key."
Helvetica Bold	Bold Helvetica text indicates an option to choose or text to type. It usually appears in numbered steps as shown in the following example: 1. Type **2.5** in the Line spacing entry area. 2. Click **Apply**.
Italics	Words are italicized for emphasis or to draw your attention to a new term. For example, "*Do not* press the RETURN key," or "This action is called *word wrapping*." Italic type is also used to indicate variable information, as in "Put Applixware in the /user/*your_name* directory."

Menu Name → Option Name	Whenever you see a reference to a menu option, the option is identified using the following notation: Menu Name → Option Name For example, "Choose **File** → **Save**."
OK and Apply	When numbered instructions are included in the text, we omit the final "Click **OK** or **Apply**" statement for brevity.

1 ELF Language Elements

This chapter describes the structure of the ELF (Extension Language Facility) language. All topics are arranged in alphabetical order. These topics are divided into four groups, as follows:

Macro Constructs - comments, constants, DEFINE statement, ENDMACRO statement, EXTERN statement, FUNCTION statement, INCLUDE statement, MACRO statement, UIMACRO statement

Variables and Data - ARRAYOF, arrays, data types, FORMAT, global variables, local variables, string variables, system variables, VAR statement

Operators - AND, arithmetic operators, EQV, IMP, logical operators, NOT, OR, relational operators, XOR

Program Control - BREAK statement, Case statement, conditional statements, For loops, GOTO statement, IF statement, NEXT STEP, ON ERROR statement, ON GOTO statement, RETURN statement, WEND statement, WHILE loops

AND Operator

AND is both a logical operator and a bitwise operator.

As a logical operator, AND is used primarily in conditional expressions. For an expression containing an AND operator to evaluate to TRUE, both of the objects in the expression must be TRUE.

AND can also be used for bitwise operations. For example, the following statement displays the number 4 in an information box.

```
info_message@(5 AND 6)  ' 0101 AND 0110 = 0100
```

The following example is an IF statement that tests an expression, and executes two statements if the tested expression is TRUE.

```
IF Buy_Applix_Stock AND Stock_Splits
{
Bank_Account = Bank_Account + 1000000
Goto EARLY_RETIREMENT
}
```

Refer to the section "OR Operator" for related information.

Arithmetic Operators

An arithmetic operator is the part of an expression that specifies the type of operation or calculation to be performed. The expression Budget + .25 instructs ELF to add the numeric constant .25 and the

value of the variable Budget. The addition sign (+) is an example of an ELF arithmetic operator.

ELF supports arithmetic, relational, and logical operators. The following tables list all ELF arithmetic operators. The column "Leading/Trailing Spaces" indicates whether you are required to include spaces before and after the operator when you type the operator.

Table 1-1. Arithmetic Operators

Arithmetic operator	Operation	Example	Leading/Trailing Spaces
-	Negative	-450	optional
^	Exponentiation	2^5	optional
*	Multiplication	50*2	optional
/	Division	100/2	optional
\I	Integer division	100\2	optional
MOD	Remainder operator	22 MOD 4	required
+	Addition	5+5	optional
-	Subtraction	10-2	optional
++	String concatenation	Area code++ Phone	option

ARRAYOF Statement

To declare a variable to be an array of FORMAT variables, you must use the ARRAYOF statement. The format for the ARRAYOF statement is as follows:

```
VAR FORMAT ARRAYOF Formatname Variablename
```

The following code sample uses a FORMAT variable called budget_info. The variable Years is established as a formatted array containing the fields in the budget_info format.

```
FORMAT budget_info
        first_qtr_profit,
        second_qtr_profit,
        third_qtr_profit,
        fourth_qtr_profit,
        year_profit
/*
*       Formats are declared before the start of the macro.
*/
get budgets

var x
var Format Arrayof budget_info Years

/*
*       This declaration declares the variable Years to be
*       a formatted array containing the fields shown in the
*       budget_info format.
*/

for x = 0 to 5
        Years[x].first_qtr_profit = 1000
next x

/*
*       You can use dot notation to access elements for the
*       formatted array.
*/

        return(Years[4].first_qtr_profit)
endget
```

Arrays

You can use an ELF array to store a collection of data. Arrays in ELF are *heterogenous,* which means that the data in a single array can contain any combination of data types. An array is not limited to data of only one type. The following code fragment defines an array with three different types of data:

```
set  heterogenous_array
var x
x = "hello", 1, NULL          ' The array contains three elements
endset
```

Declaring an Array

In ELF, an array is declared using a VAR statement. You do not declare the size of an ELF array. The size of the array is maintained automatically by ELF according to the number of elements that you assign to the array. For example, in the following macro, three names are assigned:

```
set Bears
var bears
bears = "Mama", "Papa", "Baby"        ' bears contains three elements
endset
```

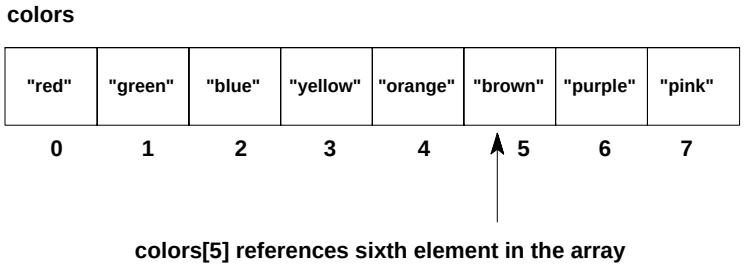
The name of the array must be unique within the method or macro.

Referencing Array Elements

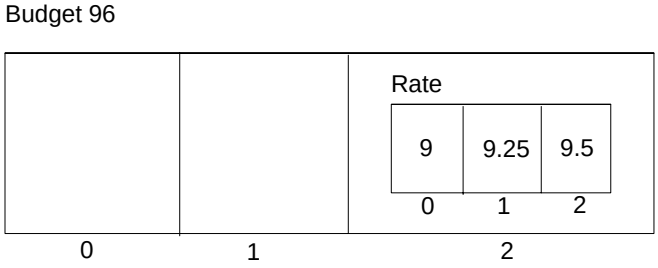
To reference the individual elements of an array, use the following notation:

```
array_name[element_number]
```

array_name is the name you give the array. element_number indicates the location of the element within the array. ELF arrays are zero-based, meaning the first array position is location 0, the second array position is location 1, and so on. The following shows the notation you use to reference the sixth element in the array named colors.



You can include an array as the element of an array. Arrays referenced by other arrays are called *sub-arrays*. In this sub-array example, the Budget96[2] array element contains the sub-array Rate.



You can reference a value contained in a sub-array directly using the element number that corresponds to the value's position in the sub-array. The format for referencing sub-array elements is:

```
array_name[element_number, sub_array_element,
                    sub_array_element...]
```

For example, Budget96[2,0] references the value 9, Budget96[2,1] references the value 9.25, and Budget96[2,2] references the value 9.5 as shown in the following figure.

Arrays can have as many elements as you need. Any element can also contain an array. In this way, you can easily create large multi-dimensioned arrays.

Assigning Values to Arrays

You use the following statement to assign a value to an array element:

```
array_name[element_number]=expression
```

array_name is the name of the array variable. element_number is the position within an array that the value will occupy. The expression is the value to be assigned to the element.

You can assign string values, number values, or other arrays to an array element. For example, the following assignment statements create an ELF array, Name, containing six string values:

```
Name[0]="Smith"
Name[1]="Jones"
Name[2]="MacIntosh"
Name[3]="McCabe"
```

```
Name[4]="Stone"
Name[5]="Vito"
```

Short Form Array Assignments

You can also create the same array using the following short form:

```
array_name=expression, expression, expression...
```

For example, the Name array could be assigned as follows:

```
Name="Smith", "Jones", "MacIntosh", "McCabe", "Stone", "Vito"
```

In this short form array assignment, array values are separated by commas. A value's position within the assignment statement determines its position within an array.

You can assign values to sub-arrays in either of two ways:

- Assign values to the sub-array and then assign the sub-array to the array. For example, if Rate is a sub-array contained in the array named Budget96, you could assign values as follows:

  ```
  Rate = 9, 9.25, 9.5
  Budget96[2] = Rate
  ```

- Assign values to sub-arrays elements directly by referencing the element in your assignment statement. For example, if Rate is a sub-array contained in the array named Budget96, you could assign values as follows:

  ```
  Budget96[2,0] = 9
  Budget96[2,1] = 9.25
  Budget96[2,2] = 9.5
  ```

Short Form Array Assignments Using Braces

ELF also allows you to indicate an array element using braces. for example, the last Name declaration could also be written as follows:

```
Name={"Smith", "Jones", "MacIntosh", "McCabe", "Stone", "Vito"}
```

In this case, the braces add no value. Here are a few examples that show where they should be used.

- When passing variables to a function, braces are easier than packing information into a temporary value. For example, the following two methods perform the same action:

```
set TEST1(arg1, arg2)
var packing
packing = arg1, arg2
DO_SOMETHING(packing)
endset
set TEST2(arg1, arg2)
    DO_SOMETHING( {arg1, arg2} )
endset
```

- Braces let you indicate multi-dimensional data. For example:

```
data = {"The", { "quick", "brown"}, "fox"}
```

  This statement creates a three-element array. The second element is also an array containing two elements.

  In the following example, a variable with one array element is created:

```
data = {1}
```

Although data has only element, ELF interprets it as an array.

In some cases, you need to remove the data from an array but need to keep the variable's type as an array. You can do this in one of two ways:

```
data = TRUNC_ARRAY@(data, 0)
data = {}
```

Refer to the "FORMAT Statement" section for related information.

BREAK Statement

The BREAK statement tells ELF to stop processing the loop and resume execution with the statement following the FOR loop. The syntax for this statement is:

```
BREAK var
```

where var is the counter variable of the FOR loop. var is optional. If it is omitted, BREAK stops processing for the current loop. The following example shows a simple BREAK statement, with no argument.

```
FOR i = 0 to 99
        ...
        if COND = -1
                BREAK           'The Loop is terminated
        ...
NEXT i
/*
*       Execution Resumes Here
*/
```

The following example shows two loops. When the inside loop reaches 50, the BREAK statement terminates both loops.

```
FOR outside = 0 to 99
```

```
        FOR inside = 1 to 99
                IF inside = 50
                        BREAK outside
        NEXT inside
NEXT outside
/*
*       Execution resumes here. Note that a simple BREAK
*       statement would have resumed with the NEXT outside
*       statement. By specifying outside in the BREAK statement,
*       both loops are terminated.
*/
```

CASE Statement

The CASE statement is a conditional statement that tests an expression against a set of possible values for that expression. A CASE statement is formatted as follows:

```
CASE OF expression
CASE value1 [, value2]...
    statement
    statement
    ...
CASE value3 [, value4]...
    statement
    statement
.
.
.
[ DEFAULT
    statement
    ... ]
ENDCASE
```

The expressions in the CASE statement, like all ELF expressions, can either be numeric or string. However, in one CASE statement all expressions must either be numeric or string.

The DEFAULT keyword and the statements that follow it are optional.

```
CASE OF result          ' Result is a string in this example
CASE "S", "T"
    menu_demo1
CASE "F"
    menu_demo2
CASE "V"
    menu_demo3
DEFAULT
    menu_demo4          ' If result does not meet any of the
                        ' previously-specified values.
ENDCASE
```

All statements following the CASE statement are executed until one of the following is encountered:

- Another CASE statement
- A DEFAULT statement
- An ENDCASE statement

After encountering one of these statements, ELF resumes execution with the statement that immediately follows the ENDCASE statement.

Comments

A comment is text in your object method source that is ignored when the source is compiled and executed. ELF supports two methods for placing comments in your object method source:

- You can comment text on a single line by placing a single quotation mark (apostrophe) before the comment text as shown:

```
set PrintSales        'This macro prints on the Sales
                      ' Dept's  laser printer.
    VAR copies
    copies = PROMPT@("How many copies?")
    ...
```

  Any text that follows the quotation mark on the same line is considered a comment and is ignored.

- You can comment large blocks of text on multiple lines using the /* and */ comment symbols. With this method, the symbol /* signals the start of a comment and the symbol */ signals the end of the comment. All text between the two symbols is recognized as a comment. For example:

```
set LoadASCII
 /*
* This macro opens an operating system file, reads it, and
* writes the contents to a Words document.
*/
    VAR op_file, name
    ...
```

Conditional Statements

ELF provides two methods that allow you to execute groups of statements based on a condition: the CASE statement and the IF statement.

Refer to the "CASE Statement, and IF Statement" sections for related information.

Constants

Constants are numerical or string values that do not change throughout an ELF program. ELF provides three constants: TRUE, FALSE, and NULL.

The constant TRUE has the value -1. The constant FALSE has the value 0. The constant NULL is the null datum value.

TRUE and FALSE can be used interchangeably with -1 and 0 in any of your methods or functions. For instance, the following statement tests whether a toggle button is toggled on (value = -1).

```
IF DB_CTRL_GET_VALUE@(dbox,"Toggle1") = TRUE
```

When the return value of a method is TRUE or FALSE, the method returns -1 (TRUE) or 0 (FALSE). Therefore, if you have a statement that displays the return value of such a method, the value -1 or 0 is displayed. For example, suppose your method includes the following statements for returning the value of a toggle button in a dialog box:

```
VAR value
value = DB_CTRL_GET_VALUE@(dbox, "Toggle1")
INFO_MESSAGE@("The value of toggle 1 is" ++value)
```

If value is determined to be TRUE, the INFO_MESSAGE@ macro displays the following in a dialog box:

```
The value of toggle 1 is -1
```

User-Defined Constants

To assign a name to a constant value, use the DEFINE statement. After you assign a name to a constant, you can use the name in place of the constant. This can enhance the readability of your code.

The format of the DEFINE statement is:

```
DEFINE name value
```

For more information, see the section "DEFINE Statement", earlier in this chapter.

Data Types

ELF supports five types of data: numbers, strings, arrays, and binary objects, and NULL.

Numbers

Numbers are stored in ELF as 16-digit, double-precision, floating-point values.

If a number is larger than six digits, ELF displays it in scientific notation. Arithmetic calculations are performed with full precision. When a number is converted to a string, the converted string has a maximum of six digits of precision.

When specifying numbers, you can use regular notation or scientific notation. For example, you can specify a number as 290000000 or as 2.9e+8.

You can use IS_NUMBER@ to determine if the value of a variable is a number.

Strings

An ELF string value is a sequence of ASCII characters. The only constraint on the size of a string is the amount of memory that exists on your system. String values must always be enclosed in double quotes as in the following example:

```
name = "string"
```

You can include double quotes within a string by preceding them with a back slash (\). For example:

```
"the answer \"no\" is correct"
```

You can include a back slash in a string by preceding it with a back slash (\\).

You can use IS_STRING@ to determine if the value of a variable is a string.

Refer to the section "String Variables" for more information.

New Line Notation

You can indicate that a new line should be inserted in an INFO_MESSAGE@ string using the notation "\n" to specify the new line. When a new line symbol is encountered in a string used by INFO_MESSAGE@, a new line is inserted and the remainder of the string is placed on the next line.

For example, the macro

```
INFO_MESSAGE@("Name is invalid\nPlease type another
              name")
```

displays the following message in a dialog box:

```
Name is invalid
Please type another name
```

Tabs and Spaces

To add special characters to a string, you can enter the character in your string by using the following notation:

\(*num*)

where *num* is the ASCII representation of the string. A tab character, in this notation, is an ASCII 9, and a space character is an ASCII 32. For example:

```
set test
    info_message@("hello\(9)\(9)hello")      ' Two tabs are added
end set
```

Arrays

An ELF array is a group of elements consisting of numbers, strings or other arrays. An array has no fixed size. The array size increases as you assign values to the array. An ELF array can contain as many elements as memory can hold.

Arrays in ELF are *heterogenous*. A heterogenous array can contain more than one type of data. For example, the following code sample defines an array that contains both numbers and strings:

```
set array_test
    var array_variable
    array_variable = 1, "two", "three", 4
    dump_array@(array_variable)
endset
```

ELF arrays are zero-based. This means that the value that occupies the first position in an ELF array is in position 0. In the array_test macro, array_variable[0] = 1, array_variable[1] = "two", and so on.

You can use IS_ARRAY@ to determine if a variable is an array. Use ARRAY_SIZE@ to determine the current size of an array.

Binary Objects

A binary object contains bytes of data. Binary objects are useful for representing information that is manipulated or analyzed byte by byte. For example, you can convert a file to a binary object for transfer to a machine that might not be able to interpret other file formats.

You can use IS_BINARY@ to determine if the value of a variable is a binary object. There are also a collection of macros for interpreting and manipulating binary data.

Null Datums

The null datum is the value of a variable that has not been assigned a value. With a macro call that accepts optional arguments, any argument that is not supplied has the value of the null datum. The pre-defined constant NULL is used to represent the null datum.

You can use IS_NULL@ to determine if the value of a variable is NULL.

DEFINE Statement

The DEFINE statement is used to assign names to constants. This often enhances the readability of your code. The format of the DEFINE statement is:

```
DEFINE name value
```

For example, you could assign the name OFF to the constant value 0 as follows:

```
DEFINE OFF 0
```

Now, whenever the name OFF is used in your macro, it is interpreted by ELF to be the value 0. For example:

```
VAR switch_set
switch_set = OFF
```

The following macro uses two DEFINE statements.

```
DEFINE Children 2
DEFINE Income 20000

set College

var money

/*
*     The Random_Seed@ macro generates a random integer from 0 to 32767
*/
      money = Random_Seed@()+ Income
/*
*     The two IF statements below use a variable (money) and a constant (children)
*/

      if money/children > 20000 then info_message@("You are going to college!")

      if money/children < 20000 then info_message@("You need a scholar ship!")
endset
```

DEFINE statements are often used in ELF header files to assign names to error codes, such as:

```
DEFINE ERR#WRONG_WINDOW 206
```

DEFINE statements are usually placed at the start of the object method source, before any methods or macros. The DEFINE statements only apply to the object method source in which they reside.

DEFINE statements are often put in a header file, and included in an object method source through the INCLUDE statement.

Refer to the sections "INCLUDE Statement" and "Constants" for related information.

ENDMACRO Statement

The Endmacro statement is the last statement in an ELF macro. ENDMACRO takes no arguments.

Refer to the section "MACRO Statement" for related information.

EQV Operator

EQV is both a logical operator and a bitwise operator.

As a logical operator, EQV is used primarily in conditional expressions. For an expression containing an EQV operator to evaluate to TRUE, both of the expressions must be either TRUE or FALSE. A truth table for EQV operator follows:

EQV	**0**	**1**
0	1	0
1	0	1

Note that:

```
EQV = Not XOR
```

EQV can also be used for bitwise operations. For example, the following statement displays the number -4 in an info box.

```
info_message@(5 EQV 6)  '00000101 AND 00000110 = 11111011 =-4
```

Refer to the section "Operator Precedence" for related information.

EXTERN Statement

Use the EXTERN statement to reference a global variable that is used in the current macro document, but is declared in a different installed macro document.

For example, if you declare the global variable Inflation in the macro document named Budget and then use the Inflation variable in the macro document named Projections, you must reference the Inflation variable in the Projections document using an EXTERN statement.

An EXTERN statement can be placed at any point in a macro document, but it must appear prior to any statement that references the global variable. Once a global variable has been referenced using an EXTERN statement, any macro in the macro document can reference that variable.

The format for an EXTERN statement is:

```
EXTERN variable_name[, variable_name, variable_name...]
```

For example, in the following statement the global variables Orders, Inventory, and Units are referenced:

```
EXTERN Orders, Inventory, Units
```

FOR Loop

The FOR and NEXT statements create a FOR loop that executes a series of statements a specified number of times.

The format of a FOR loop is:

```
FOR variable=expression TO stop_expression [STEP
        expression]
loop statements
NEXT variable
```

A FOR loop is a convenient method for loading data into an array. In the following method, the numbers 1 to 100 are loaded into the Numbers array.

```
set  loadarray
var loop_counter, Numbers

for loop_counter = 1 to 100                ' loop counter is set to 1 for the first pass
                                           ' through the loop, then incremented by
                                           ' 1 for each successive pass. In this example, the
                                           ' loop will be executed 100 times.

/*
*     loop_counter provides the index to the Numbers array and the value loaded into
*     the array.
*/

Numbers[loop_counter] = loop_counter

next loop_counter                          ' The NEXT provides an ending statement for
                                           ' the FOR loop.

dump_array@(Numbers)          ' Displays the contents of the numbers array
endset
```

When ELF encounters a FOR statement, it assigns the value of an expression to a variable. It then compares this variable value to a stop

expression. If the variable value is greater than the stop value, ELF resumes execution of the method at the line following the NEXT statement. If the variable value is less than the stop value, ELF executes the loop statements.

When ELF encounters the NEXT statement, it increments the variable by the STEP value. The default STEP value is 1. ELF then returns to the FOR statement and re-evaluates it. The next section describes the STEP statement in more detail.

STEP Statement

You can include an optional STEP value if you do not want to use the default value of 1. If you provide a negative STEP value, the loop statements are performed until the variable value is less than the stop value. The following code has two FOR loops. The first loop loads the even numbers from 1 to 100 into the Numbers array. The second loads the even negative numbers from 0 to -100 into the Numbers array.

```
set  loadarray
var loop_counter, Numbers, x
/*
*   Loop counter is set to 2 for the first pass through the loop, then incremented
*   by 2 for each successive pass. In this example, the loop will be executed
*   50 times.
*/
for loop_counter = 2 to 100 STEP 2    '
    x = x + 1                                   ' x is used as the array index.
    Numbers[x] = loop_counter

next loop_counter                                       ' The NEXT provides an ending statement
                                                ' for the FOR loop.

for loop_counter = -2 to -100 STEP -2
/*
*   Loop counter is set to -2 for the first pass through the loop, then decremented
*   by 2 for each successive pass. In this example, the loop will be executed 50
*   times.
```

```
*/
x = x + 1                                   ' x is used as the array index.
Numbers[x] = loop_counter
next loop_counter                           ' The NEXT provides an ending statement for the
                                            ' FOR loop.
dump_array@(numbers)
endset
```

Refer to the sections "BREAK Statement" and "NEXT STEP Statement" for related information.

FORMAT Statement

Use the FORMAT statement to define *format variables.* A format variable is a type of array variable that describes the structure of the data. Using a format variable, you can assign names to all or part of an array.

Formats are defined using the FORMAT statement as follows:

```
FORMAT format_name
    element0_name,
    element1_name,
    element2_name,
    ...
    elementN_name
```

format_name is a name that identifies the format. element_name items are the names by which you want to refer to the format's elements. Each element name in the FORMAT statement is separated by a comma.

For example, suppose you want to define a format that contains a "profit" value for each of four fiscal quarters and for the year:

```
FORMAT budget_info
    first_qtr_profit,
    second_qtr_profit,
    third_qtr_profit,
    fourth_qtr_profit,
    year_profit
```

After defining the format, you can define a variable that uses it as follows:

```
VAR FORMAT template_name variable
```

For example, you can apply the budget_info format to the variable budget96 as follows:

```
VAR FORMAT budget_info budget96
```

When a format is applied to a variable, you can use the element names defined in it to refer to array elements. For example, you could use the format names to assign values to the variable as follows:

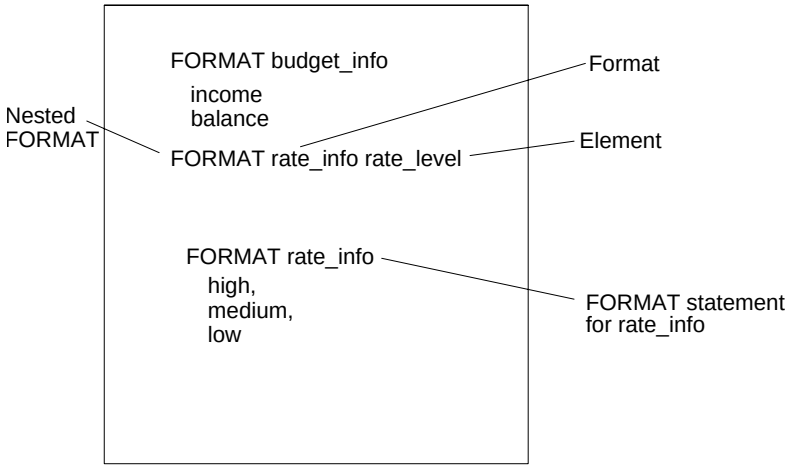
```
variable.element_name=expression
```

For example:

```
budget96.first_qtr_profit = 22534.89
budget96.second_qtr_profit = 12345.00
budget96.third_qtr_profit = -10234.90
budget96.fourth_qtr_profit = 35897.50
budget96.year_profit = 60542.49
```

Nesting FORMAT Statements

An element within a formatted variable can also be a formatted variable. That is, you can nest FORMAT statements to define sub-formats, as shown in the following:



You assign values to sub-array elements using the following format:

```
variable.element_name.sub_element_name = expression
```

For example:

```
VAR FORMAT budget_info budget96
budget96.rate_level.high = 12.5
budget96.rate_level.medium = 11.75
budget96.rate_level.low = 10.865
```

All format elements and sub-format elements can be referenced by stringing together the array variable names, each separated by a period (.), and adding the element name to the end.

Function Statement

A FUNCTION is a collection of ELF statements whose purpose is to return a numerical value. For example, you might define a FUNCTION that returns a sum of numbers.

A FUNCTION is formatted as follows:

```
FUNCTION name(arguments)
    statements
    RETURN(value)
ENDFUNCTION
```

Function Names and Arguments

The name of a FUNCTION must be unique within the ELF task, and must adhere to the following naming conventions:

- Names must begin with a letter of the alphabet.
- Names can contain alphabetic, numeric, or underscore (_) characters.
- Names cannot include spaces.

Function names must not include:

- the "#" character.
- The @ symbol, $ symbol, or _ (underscore) symbol as the last character in the macro name. By convention, only ELF built-in macros can use these symbols as the last character in the macro name.

The arguments are an optional list of arguments that are passed to the function.

Recording Functions

A call to a function is not recorded by the Keystroke Recorder. This is the only important difference between a FUNCTION and a standard macro.

By convention, a function returns a number to the calling macro, but ELF does not enforce this convention.

Refer to the section "RETURN Statement" for additional information on returning information to a calling macro.

The following code shows a simple macro that calls a function.

```
macro Call_function
var x, total, add_to_total, function_argument

for x = 1 to 10
      function_argument = x
      add_to_total = Double_It(function_argument)        ' Calls the Double_It function
      total = total + add_to_total
next x

info_message@(total)                                     ' The total is 110.
endmacro

/*
*     The Double_it function accepts a  number from the calling macro, doubles it,
*     and returns the result.
*/
Function Double_It(x)

x = 2*x
return(x)

endfunction
```

Global Variables

Global variables are defined independently of ELF macros and may be referenced by any macro within a given task. A global variable retains its value for the duration of a task.

Global variables are declared using VAR declarations that are placed at the start of the macro document, outside of any macro or function. For example, the following declares Inflation as a global variable:

```
VAR Inflation
MACRO Budget90
    Inflation = 5
    ....
    statements
    ....
ENDMACRO
```

When a task is initiated, the value of all global variables in the task is the null datum.

Use EXTERN to reference a global variable that is used in the current macro document, but is declared in a different installed macro document.

Global variables are only global to a single ELF task. If you want to exchange variable information between ELF tasks use a system variable.

Refer to the sections "System Variables" and "EXTERN Statement" for related information.

GOTO Statement

A GOTO statement is used to branch to a specific location in your document. The format of a GOTO statement is:

```
GOTO label
```

The label is a text string that appears directly before the statement you wish to branch to. The label is followed by a colon. The following example shows the GOTO statement.

```
set Beggar
var answer
ask_again:
    answer = prompt@("Can I borrow a quarter?")
    if answer <> "yes" Goto ask_again
endset
```

IF Statement

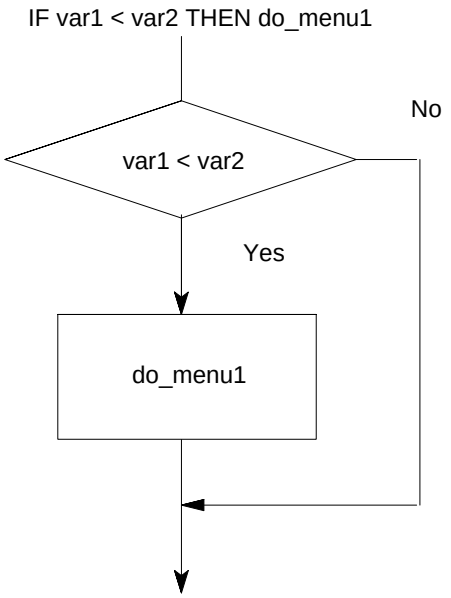
IF-THEN and IF-THEN-ELSE statements specify what statements in an ELF macro are executed based on the evaluation of an expression.

An IF-THEN statement is formatted as follows:

```
IF expression THEN statement
```

If the value of expression is not FALSE (a value other than 0), ELF executes the statement. If the value of expression is FALSE, then the statement is not executed and processing continues with the next statement after the IF-THEN statement.

The expression can compare values, such as IF var1 > var2, or it can be a value, such as IF var. The keyword THEN is optional. The following shows an IF-THEN statement:



In the following example, ELF throws an error if the expression (column < 0) evaluates to TRUE. If the expression evaluates to FALSE, ELF continues processing at the statement IF (column > 26).

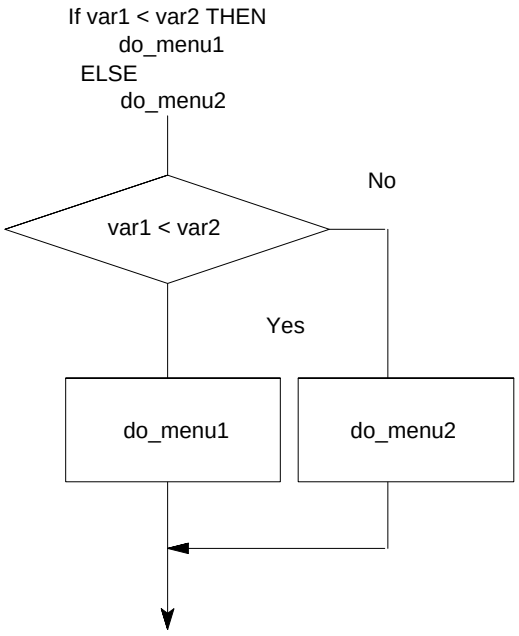
```
IF (column < 0)
    ERROR@(99, "Not a valid column")
IF (column > 26) ...
```

IF - THEN - ELSE

An IF-THEN-ELSE statement is formatted as follows:

```
IF expression THEN statement1 ELSE statement2
```

With an IF-THEN-ELSE statement, statement1 is executed if expression evaluates to TRUE. Otherwise, statement2 is executed. The following shows an IF-THEN-ELSE statement.



The ELSE operation in an IF-THEN-ELSE statement is typically another IF-THEN-ELSE statement, as shown in the following example:

```
IF result = "S"
    menu_demo1
ELSE IF result = "F"
    menu_demo2
ELSE IF result = "V"
    menu_demo4
```

See the sections "CASE Statement" and "Conditional Statements" for related information.

IMP Operator

IMP (implication) is both a logical operator and a bitwise operator.

As a logical operator, IMP is used primarily in conditional expressions. For an expression containing an IMP operator to evaluate to FALSE, The first expression must be TRUE, and the second expression must be FALSE. Otherwise the expression containing an IMP operator is TRUE.

A truth table for IMP operator follows:

IMP	0	1
0	1	1
1	0	1

IMP can also be used for bitwise operations. For example, the following statement displays the number -2 in an info box.

```
info_message@(5 IMP 6)  ' 00000101 AND 00000110 = 11111110 = -2
```

Refer to the section "Operator Precedence" for related information.

INCLUDE Statement

For an object method source to use information in a header file, the header file must be included in the source. The INCLUDE statement is

used to include a header file in a source The format for the INCLUDE statement is:

```
INCLUDE "filename"
```

where filename is the name of the header file you want to include. filename must be enclosed in quotation marks and should include the file name extension. For example, to include the ELF include file Words_.am, use the following line:

```
INCLUDE "words_.am"
```

When an INCLUDE statement is encountered, ELF looks for the header file in three directories, in this order:

1. Your axhome/macros directory.
2. *install_dir*/axlocal. This is your Applixware install directory.
3. *install_dir*/axdata/elf directory. This directory contains the ELF header files.

If the header file is not in any of these directories, then you must use a full path name to indicate the directory in which the header file is placed.

All INCLUDE statements should be placed at the beginning of the macro document. It is also recommended that header files contain only DEFINE, FORMAT, and EXTERN statements. If you include a pound sign (#) before your INCLUDE statement, it is ignored. For example:

```
#INCLUDE "wp_.am"     ' This line works - the # is ignored.
```

NOTE: ELF does not support nested include files. INCLUDE statements in ELF include files cause compiler errors.

Local Variables

A local variable is defined within an ELF macro or method and can only be referenced by that macro or method. To declare a local variable, you include the VAR statement between the opening statement of the macro or method (MACRO, UIMACRO, FUNCTION, get, set) and the terminating statement of the macro or method (ENDMACRO, ENDFUNCTION, endget, endset). The following example declares the local variables contents and file.

```
FUNCTION Ascii
    VAR contents, file

    file =prompt@ ("Name of file")
    contents=Read_Ascii_File@(file)
    RETURN(contents)
ENDMACRO
```

When a macro or method is called, all locally declared variables have the null datum as an initial value.

Refer to the sections "Variables", "Global Variables" and "System Variables" for related information.

Logical Operators

An operator is the part of an expression that specifies the type of operation or calculation to be performed. Logical operators are used for two purposes in ELF:

- To compare two expressions.
- Perform bitwise operations, creating a numeric value from two numeric values.

The following expression contains a logical operator:

```
if X =10 and Y = 5  statement
```

The expression instructs ELF to compare the variable X with 10, and compare the variable Y with 5. and act on the ensuing statement if both X = 10 and Y = 5 are TRUE. The word and is a logical operator.

All of the ELF logical operators can be used to perform bitwise operations on numeric values. For example:

```
if X and Y then statement
```

where X and Y are numbers. The corresponding bit of each number are compared. If both corresponding bits are 1, then the solution contains a 1 in that location. For example:

```
              0 0 0 0 0 1 1 0     ' The number 6 in binary
AND           0 0 0 0 0 1 0 1     ' The number 5 in binary
-------------------------------------------------------------------------
                0 0 0 0 0 1 0 0     ' The number 4 in binary
```

Therefore,

```
5 AND 6 = 4          ' Bitwise AND
```

ELF supports arithmetic, relational, and logical operators. The following table lists all the ELF logical operators. The column "Leading/Trailing Spaces" indicates whether you are required to include spaces before and after the operator.

Table 1-2 Logical Operators

Logical operator	Operation	Example	Leading/Trailing Spaces
Not	Inverse	NOT a=c	Required
AND	And	a>b AND a>c	Required
OR	Or	a>b OR a>c	Required
XOR	Exclusive or	a>b XOR a>c	Required
EQV	Equivalence	a-b EQV a-c	Required
IMP	Implication	a-b IMP a-c	Required

Refer to the section "Operator Precedence" for related information.

Macro Statement

The MACRO statement is the first statement in an ELF macro. The MACRO statement is always followed by a string, which is the name of the ELF macro.

Macro Names and Arguments

The name of a macro must be unique within the ELF task, and must adhere to the following naming conventions:

- Names must begin with a letter of the alphabet.
- Names can contain alphabetic, numeric, or underscore (_) characters.

- Names cannot include spaces.

Variable and macro names must not include:

- the "#" character.
- The @ symbol, $ symbol, or _ (underscore) symbol as the last character in the macro name. By convention, only ELF built-in macros or methods can use these symbols as the last character in the macro name.

The arguments are an optional list of arguments that are passed to the macro. The argument list is a set of values passed from another ELF macro. If more than one argument is passed from another macro, the arguments are separated by commas:

```
macro Example(arg1, arg2)
```

The following example shows two ELF macros. Test passes a string to the macro StringLength. Note that the MACRO statement for StringLength contains the argument passed from test.

```
macro Test

var astring, length

    astring = "Hello, this is a string."
    length = StringLength(astring)          ' Call the StringLength Macro with astring
                                                ' as an argument.
    info_message@("The length is "++length)
endmacro

macro StringLength(str)
/*
*   The str variable does not have to be declared here because
*   it was declared in the test macro.
*/

    return(len@(str))    ' Return the length of the string

endmacro
```

Refer to the section "ENDMACRO Statement" for related information.

NEXT STEP Statement

The NEXT STEP statements tells ELF to skip the remainder of the statements within the loop's body and resume execution at the top of the FOR loop. The syntax for this statement is:

```
NEXT STEP variable
```

where variable is the counter variable of the loop. The following example shows the NEXT STEP statement.

```
FOR i = 0 to 99
    ...
    if COND = 1
        NEXT STEP i

/*
*   If COND = 1, these statements are skipped.
*/
    info_message@("Condition is not 1")    ' This statement executes if COND <> 1

NEXT i      'Execution resumes here of COND = 1. i is incremented, and the
            ' Loop continues.
```

NOT Operator

NOT is both a logical operator and a bitwise operator.

As a logical operator, NOT is used primarily in conditional expressions. For an expression containing a NOT operator to evaluate to TRUE, the object in the expression must be FALSE.

NOT can also be used for bitwise operations. For example, the following statement displays the number -7 in a message box.

```
info_message@(NOT 6)  ' NOT 00000110 = 11111001 = -7
```

The following example is an IF statement using a NOT operator. The GOTO executes if the expression x=y is FALSE.

```
IF NOT x = y
{
Goto NextCompare
}
```

Refer to the section "AND Operator" for related information.

NOTHING Statement

The NOTHING statement performs no operation. It can be used in an IF statement as shown:

```
IF answer = "no"
    NOTHING
ELSE
    info_message@("Thank You")
```

In this case, ELF executes a statement only if an expression evaluates to FALSE.

The semi-colon (;) symbol can be used in place of the NOTHING statement. For example:

```
IF answer = "no"
    ;
ELSE
    info_message@("Thank you!")
```

ON ERROR Statement

The ON ERROR statement loads a new error handler onto the execution stack of the current macro. Error handlers are run when an error is thrown by the ELF macro.

The following sample code shows the structure of an ELF error handler. This error handler checks whether the error code is 45, and if it is, the error is displayed in an error message box and processing is sent to the again label. If the error is not error code 45, it is rethrown so that another error handler or the ELF error handler can handle the error.

```
ON ERROR
{
   IF ERROR_NUMBER@() = 45
   {
            ERROR_BOX@
            GOTO again
   }
   ELSE
            RETHROW_ERROR@
}
```

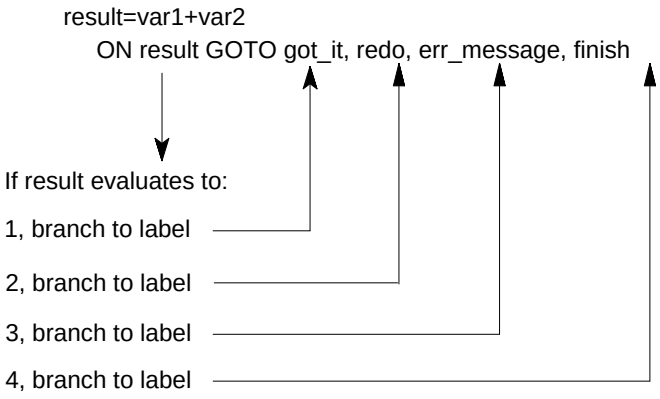
For more information about error handling, see Chapter 5, "Error Handling," in this manual.

ON GOTO Statement

The ON GOTO statement causes your method to branch to a label indicated by the computed value of an expression. The format of a computed GOTO statement is:

```
ON expression GOTO label, label, label...
```

When ELF encounters a computed GOTO statement, it evaluates the expression and then branches to the label that occupies the position in the GOTO statement list corresponding to the computed value. The GOTO statement list is 1-based. The following figure illustrates the way in which computed GOTO statements work.



For example, if the computed value of an expression is 3, ELF branches to the third label listed in the GOTO statement. In the following statement, ELF would branch to the label err_message since the value of the expression is 3:

```
result=3
ON result GOTO got_it, re_do, err_message, finish
.....
got_it:
        statements
        ...
re_do:
        statements
        ...
err_message:
        statements
        ...
finish:
        statements
        ...
```

If the computed value of an expression is 0, or the value exceeds the number of labels in the GOTO statement list, ELF does not branch out of the regular statement sequence; processing continues with the next statement following the ON GOTO statement. If the value is less than 0 or greater than 255, ELF throws an error.

If possible, you should avoid using computed GOTO statements. It is often difficult to follow the flow of a program when computed GOTO statements are used, so maintaining macros or methods that contain such statements can be cumbersome.

Operator Precedence

The order in which operations in an ELF expression are performed is determined by rules of precedence. These rules are:

- Macro or method calls are performed first
- Arithmetic operations are performed second, in the following order:

  1. arithmetic negation
  2. exponentiation
  3. multiplication or division (in order from left to right)
  4. integer division
  5. remainder operation (MOD)
  6. string concatenation
  7. addition or subtraction (in order from left to right)

- Relational operations are performed third. All relational operations share the same order of precedence.
- Logical operations are performed last, in the following order:

  1. Inverse (NOT)

  2. All other logical operations share the same order of precedence.

Operations that share the same order of precedence are performed in the order in which they are encountered (left to right).

You can change the order in which operations are performed using parentheses. Expressions contained in parentheses are evaluated first starting with the innermost parentheses. For example:

```
2+3*4+5=19
2+(3*4)+5=19
(2+3)*(4+5)=45
```

OR Operator

OR is both a logical operator and a bitwise operator. As a logical operator, OR is used primarily in conditional expressions. For an expression containing an OR operator to evaluate to TRUE, either of the objects in the expression must be TRUE.

OR can also be used for bitwise operations. For example, the following statement displays the number 7 in an information box.

```
info_message@(5 AND 6)  ' 0101 OR 0110 = 0111
```

The following example is an IF statement that tests an expression, and executes two statements if the tested expression is TRUE.

```
IF Buy_Applixware OR Buy_Applix_Stock
{
Bank_Account = Bank_Account + 1000000
Goto EARLY_RETIREMENT
}
```

Refer to the "AND Operator" section for related information.

Reserved Words

The following reserved words cannot be used as macros, functions, labels, or variable names:

AND	ARRAY	ARRAYOF	ARRAYOF2
BEGIN	BREAK	CALL	CASE
CONTROL	DEFAULT	DEFINE	#DEFINE
ELSE	END	ENDCASE	ENDFUNCTION
ENDGET	ENDMACRO	ENDSELECT	ENDSET
EQV	ERROR	EXTERN	FALSE
FOR	FORMAT	FUNCTION	GET
GOTO	HELP	IF	IMP
INCLUDE	#INCLUDE	INTEGER	LET
MACRO	MOD	NEXT	NODEBUG
NOT	NOTHING	NRMACRO	NULL
NUMBER	OBJECT	OBJVAR	OF
ON	OR	POINTER	RETURN
SELECT	SELECT1	SET	STEP
STRING	STRUCT	SYSVAR	THEN
TO	TRUE	UIMACRO	VALUES
VAR	WEND	WHILE	XOR
@@@			

Relational Operators

An operator is the part of an expression that specifies the type of operation or calculation to be performed. Relational operators instruct ELF to compare two values. The following expression contains a relational operator:

```
if X < 5 statement
```

The expression if X < 5 instructs ELF to compare the variable X with 5, and act on the ensuing statement if X is less than 5. The less than sign (<) is a relational operator.

ELF supports arithmetic, relational, and logical operators. The following table lists all the ELF relational operators. The column "Leading/Trailing Spaces" indicates whether you are required to include spaces before and after the operator when you type the operator.

Table 1-3 Relational Operators

Relational operator	Operation	Example	Leading/Trailing Spaces
<	Less than	1<2	Optional
>	Greater than	2>1	Optional
<=	Less than or equal	1<=2	Optional
>=	Greater than or equal	2>=1	Optional
=	Equal to	2=2	Optional
<>	Not equal to	1<>2	Optional
<<	Less than (string)	"Arthur"<<"Barnum"	Optional
>>	Greater than (string)	"Handley">>"Evans"	Optional

Table 1-3 Relational Operators (cont.)

Relational operator	Operation	Example	Leading/Trailing Spaces
<<=	Less than or equal (string)	"Clark"<<="Evans"	Optional
>>=	Greater than or equal (string)	"Evans">>="Clark"	Optional

Return Statement

When RETURN is encountered, the method, macro, or function is terminated and no other statements in the code are executed. RETURN statements are only necessary when you want to exit a macro or method before the usual processing is completed. For example, your method might include an IF-THEN statement in which the method is terminated if a certain condition is met. If no RETURN is found in a method, macro, or function, the method, macro, or function is exited when the ENDMACRO statement is encountered.

There are two forms of RETURN statements. One returns to the calling macro. The other returns a specified value.

The format of a RETURN statement that does not return a specified value is:

```
RETURN
```

The format of a RETURN statement that returns a specified value is:

```
RETURN(expression)
```

The expression is any ELF expression. Use this RETURN statement in a function that returns a value to its calling method. For example, the following method returns a string if a certain condition is met.

```
get BudgetAnalysis(CurrentVal)
    VAR actual_val, budgeted
    budgeted = PROMPT@("Enter amount budgeted for this
                item")
    actual_val = budgeted - CurrentVal
    RETURN(actual_val)
endget
```

Since, by definition, all ELF macros return a value, the RETURN statement is the same as RETURN(NULL).

String Variables

String variables are a flexible and powerful part of ELF. To assign a string to a variable, set the variable name equal to the string using double-quotes to delimit the string:

```
name = "Queen of Sheba"
```

String Concatenation

To concatenate two strings, use the ++ operator. The following method displays "Hello Applix" in an information box.

```
set Hello
    info_message@("Hello "++"Applix")
endset
```

String Compares

You can compare strings using the >> (greater than) and << (less than) operators. Characters are compared based on the ASCII value of the character. The following lists basic characters in ascending order from left to right:

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz

The following test tests two strings to see which is greater.

```
set  teststrings
var string1, string2
      string1  "First String"
      string2 = "Second String"
      if string1 >> string2
                  info_message@("The first string is greater")
      if string2 >> string1
                  info_message@("The second string is greater")
endset
```

Character Notation

To specify a special character within a string, you can express the character by its ASCII numeric value using the following notation:

\(num)

For example, the tab character is ASCII 9 and the space character is ASCII 32:

data = "The\(32)quick\(9)brown fox"

Prints as follows:

The quick brown fox

Executable String Variables

If a variable is assigned a string value, and that string is the name of an executable macro or function, the variable is executable. You indicate that a string variable should be executed by preceding the variable name with an exclamation point (!).

For example, suppose you declare a variable and assign it a string value as follows:

```
VAR check
check = "DIR_EXISTS@"
```

You can execute the check variable by preceding it with an exclamation point:

```
!check("/usr/sam")
```

In this example, !check is interpreted as DIR_EXISTS@, so the function DIR_EXISTS@("/usr/sam") is executed.

Executable string variables are useful in instances when the choice of what macro to run is dependent on whether certain conditions are met. In the following example, the method that is run depends on the response typed at a prompt.

```
set  start_app
   VAR answer, do_macro
try_again:
   answer = PROMPT@("Number of program to run")
   IF answer = "1"
      do_macro = "budget"
   ELSE IF answer = "2"
      do_macro = "transactions"
   ELSE IF answer = "3"
      do_macro = "year_to_date"
   ELSE
   {
      INFO_MESSAGE@(
         "Invalid response.  Please type another number")
```

```
        GOTO try_again
    }
    !do_macro
endset
```

In this example, the statement !do_macro indicates that the macro budget, transactions, or year_to_date is run depending on the value of the answer variable.

Refer to the section "Variables" for more information.

System Variables

System variables are variables that retain their value throughout the Applixware environment. For instance, you could assign a value to a system variable used within an object method and then use the value from the object method in a different object method. ELF supports two different system variable methods. The first declares variables in a way similar to the way you declare global variables. For example, here is the declaration of system variable named BUDGETinfo:

```
SYSVAR BUDGETinfo
```

The second method, which uses ELF macros, is discussed in the next section.

If your application is contained within more than one source file, place this declaration in each of the source files. (The best way to do this is to place the declaration in a header file, then include the header file in all of your source files.)

Unlike global variables, you do not have to use the EXTERN keyword to tell ELF that the variable is defined in another file.

After you define a system variable, it remains defined until the end of your Applixware session. That is, the memory allocated to the variable

remains in use even after the task that uses the memory stops executing.

Because the memory used by a system variable remains allocated during the current session, you should be careful not to overuse this feature by placing large amounts of data into system variables. If you need to use a system variable that contains a lot of data, you can release its memory space back to ELF by assigning the variable a NULL value when you are done with the information.

Case Sensitivity with System Variables

System variable names are case sensitive when you are referencing them with the SYSTEM_VAR@ and SET_SYSTEM_VAR@ ELF macros. System variable names declared using the SYSVAR declaration are always stored in ELF in uppercase. An example follows:

```
 SYSVAR  mySysVar                                    ' This variable is stored as MYSYSVAR -
                                                     ' all uppercase.
set Test_System_Variables
      mysysvar = 10                                  ' MYSYSVAR = 10
      set_system_var@("MYSYSVAr", 12)                ' MYSYSVAr =  12
      INFO_MESSAGE@(SYSTEM_VAR@("MYSYSVAr") ) ' This returns 12
      info_message@(SYSTEM_VAR@("MYSYSVAR"))         ' This returns 10
endset
```

You can declare system variables with the SYSVAR keyword, or through the SYSTEM_VAR@ ELF macro. The SYSTEM_VAR@ ELF macro allows you to declare system variables in mixed cases, while all variables declared with SYSVAR are uppercase.

The following lists the rules for declaring system variable with and without the SYSVAR declaration:

- sysvar foo is same as SYSVAR FOO
- sysvar foo is same as system_var@("FOO")

- sysvar foo is NOT same as system_var@("foo")

SET_SYSTEM_VAR@

SET_SYSTEM_VAR@ sets the value of a system variable. The format for the SET_SYSTEM_VAR@ macro is:

```
SET_SYSTEM_VAR@(name,value)
```

name, a string, is the name of the system variable for which you want the value. If you declare a system variable as a SYSVAR declaration, name must be all uppercase to access that system variable.

SYSTEM_VAR@

SYSTEM_VAR@ returns the value of a system variable. The format for the SYSTEM_VAR@ macro is:

```
value = SYSTEM_VAR@(name)
```

name, a string, is the name of the system variable for which you want the value. If you declare a system variable as a SYSVAR declaration, name must be all uppercase to access that system variable.

UIMACRO Statement

If you create a macro that calls a dialog box, you may want to designate the macro as a user interface macro. The difference between a standard ELF macro and a user interface macro is the way in which the macro is represented in a keystroke recording.

User interface macros start with the UIMACRO keyword. For example:

```
UIMACRO name(arguments)
    statements
ENDMACRO
```

The UIMACRO statement is the first statement in an ELF macro. The UIMACRO is always followed by a string, which is the name of the ELF macro.

Macro Names and Arguments

The name of a macro must be unique within the ELF task, and must adhere to the following naming conventions:

- Names must begin with a letter of the alphabet.
- Names can contain alphabetic, numeric, or underscore (_) characters.
- Names cannot include spaces.

Macro names must not include:

- the "#" character.
- The @ symbol, $ symbol, or _ (underscore) symbol as the last character in the macro name. By convention, only ELF built-in macros can use these symbols as the last character in the macro name.

The arguments are an optional list of arguments that are passed to the macro from the macro that calls it. If more than one argument is passed from another macro, the arguments are separated by commas:

```
UIMACRO Example(arg1, arg2)
```

Recording User Interface Macros

When you make a keystroke recording of a macro and the recording contains calls to a dialog box, the dialog box call is recorded in one of two ways:

- When the macro that calls the dialog box is a user interface macro, only the call to the macro that displays the dialog box is recorded. (In most cases, this is the macro DB_DISPLAY@.) When you replay the recording, the dialog box is displayed and you must enter information into the dialog box and exit the dialog box in order to continue replaying the recording.

- When the macro that calls the dialog box is a standard macro, calls to the macros that perform the actions specified by dialog box selections are recorded. When you replay the recording, the dialog box is *not* displayed. The same actions performed as a result of dialog box selections made at the time of the recording are performed when you replay the recording.

Except for the UIMACRO keyword and the way in which the macros are recorded, there is no difference between the way regular macros and user interface macros work or the way you create them.

All the menu options in Applixware applications that call dialog boxes are standard macros. When a recording is made involving an Applixware menu option that calls a dialog box, the underlying dialog box calls are recorded. When you replay the recordings, no dialog boxes are displayed.

There is no need to provide a distinction between regular and user interface macros when you create macros that do not involve dialog box calls. Therefore, you should define macros that do not involve dialog box calls as standard macros.

Variables

Variables are named data units that contain values. A variable may take the form of a string, number, array, macro, or function.

ELF variables must be declared before they are assigned values or referenced. ELF variables have no preset type. An ELF variable can be, at any point in time, a string, number, array, macro, or null datum. When an ELF variable is declared, its initial value is the null datum.

Scope

Scope refers to the accessibility of a variable. The scope of a variable is determined by the location of the variable declaration in an ELF macro. ELF supports local, global, and system variables. These variable types represent three different levels of scope.

- A local variable has the most limited scope. A local variable is accessible only from the macro in which it is declared. Local variables are declared inside the body of the macro itself.
- A global variable is one that can be used by any macro for the duration of a particular ELF task. If a global variable is assigned a value in one macro contained within a particular task, that value can be used by other macros that are run as part of the task.

  Global variables are made available to other macros in the same task through the EXTERN statement.
- A system variable has the widest scope available in ELF. It can be used by any ELF macro for the duration of the Applixware session. System variables are declared through the SYSVAR declaration, or through the ELF macro SYSTEM_VAR@().

Declaring Variables

All variables must be declared before they are used in a macro. You declare variables using the VAR Statement.

Assigning Values to Variables

Use assignment statements to assign a value to a variable. The value may be a number, string, array, or expression. The format of an assignment statement is:

```
variable = expression
```

For example:

```
charge = cost+(cost*.05)
```

An ELF variable assumes the data type of the value assigned to it. For example:

```
x = "12" 'x becomes a string variable
x = 12   'x becomes a numeric variable
```

For information on assigning values to array variables, see the section "Arrays" earlier in this chapter.

Refer to the sections "VAR Statement", "FORMAT Statement" and "VAR FORMAT Statement" for related information.

Passing Variables

Variables are always passed to macros and functions by reference. Because of this, constants that are passed to subroutines and functions cannot be changed In the example code that follows, the code on the left passes a constant, and throws the error "not a variable." The code on the right shows the proper approach, passing a variable instead of a constant.

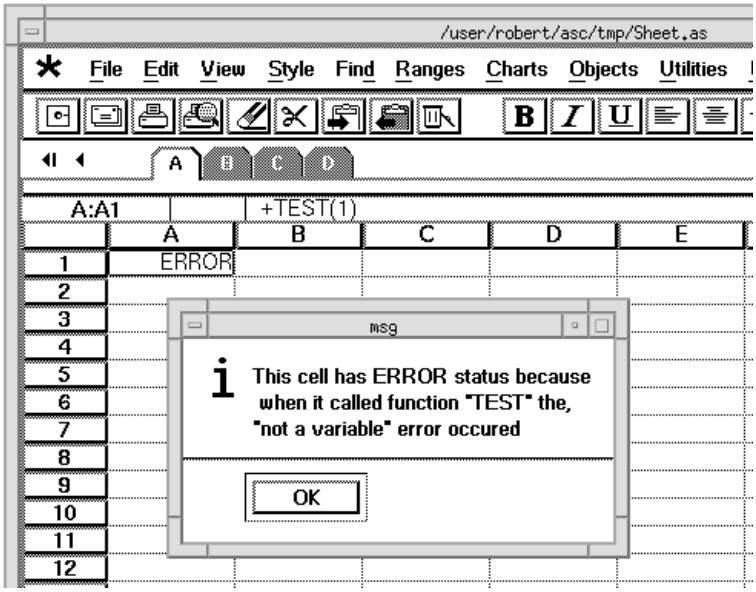
This throws an Error

```
define greeting "Hello"

macro test

    test2(greeting)

endmacro


macro test2(hello)

    hello = hello++" joe"
    info_message@(hello)
    return

endmacro
```

This Works

```
define greeting "Hello"

macro test
        var x
        x = greeting
        test2(x)
endmacro

macro test2(hello)

        hello = hello++" joe"
        info_message@(hello)
        return

endmacro
```

The "not a variable" error can also occur when you try to pass a a numeric constant from a spreadsheet to a macro that tries to change that value. For example, suppose you ran the macro on the left of the preceding table like this:



VAR Statement

All variables must be declared before they are used in a macro. You declare variables using the VAR statement. The format for a VAR statement is:

```
VAR variable_name[, variable_name]...
```

You can declare more than one variable using one variable statement and you can use multiple VAR statements within a macro. If you include multiple variable names in a single VAR statement, the variable names must be separated by commas. The VAR declaration on the left is equivalent to the three VAR statements on the right:

VAR name, address, id

```
VAR name
VAR address
VAR id
```

The location of a VAR declaration determines whether a variable is a local variable or a global variable.

A variable's name cannot be longer than 60 characters.

Refer to the section "Variables" for related information.

VAR FORMAT Statement

You can define a variable that uses a previously-established FORMAT template as follows:

```
VAR FORMAT template_name variable
```

For example, you can apply the budget_info format to the variable budget96 as follows:

```
VAR FORMAT budget_info budget96
```

Refer to the section "FORMAT Statement" for related information.

Wend Statement

The WEND statement is used to terminate a WHILE loop. The structure of a WHILE loop is as follows:

```
WHILE expression
    loop statements
```

```
WEND
```

For more information on the WEND statement, see the section entitled While Loop, later in this chapter.

Refer to the following section "While Loop" for related information.

WHILE Loops

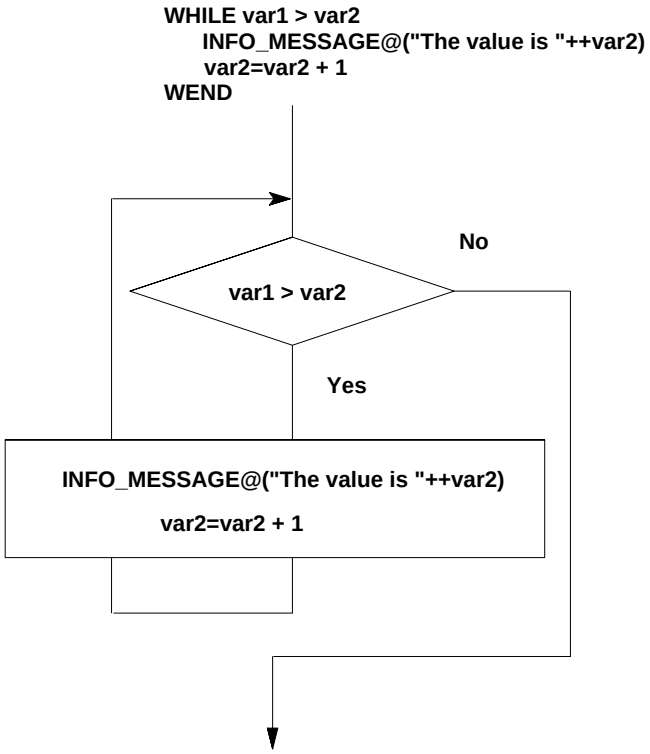
A WHILE loop executes a series of statements as long as the value of an expression is not false (a value other than 0). A WHILE loop is always terminated by a WEND statement. The format of a WHILE loop is as follows:

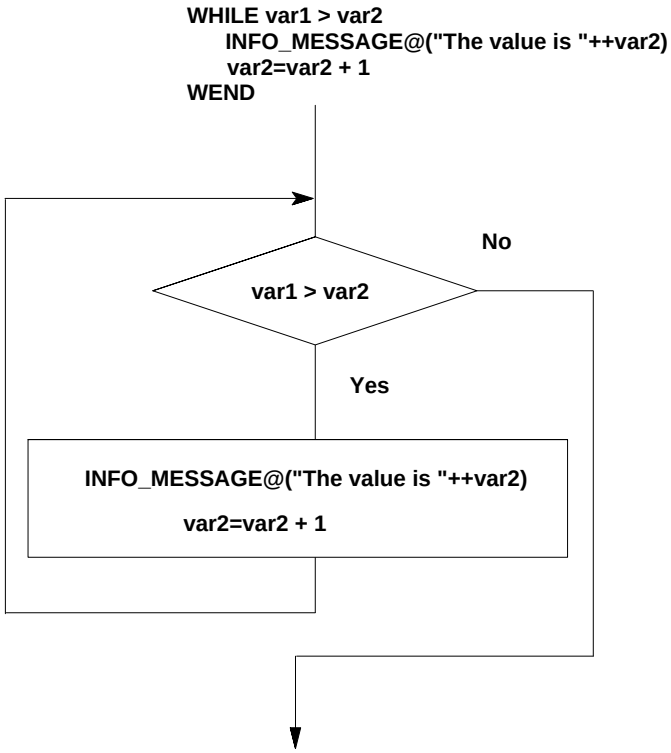
```
WHILE expression
    loop statements
WEND
```

ELF evaluates the expression following the WHILE statement. If it is not FALSE, ELF executes the statements in the WHILE loop in the order they appear.

The expression can be an expression that compares values, such as WHILE var1 > var2, or it can be a value, such as WHILE var. When ELF encounters the WEND statement, it returns to the corresponding WHILE statement and re-evaluates the expression. If the expression remains not FALSE, the process is repeated. If the expression is evaluated to FALSE, ELF proceeds to the statement following the WEND statement.

The following illustrates the processing of a WHILE loop.





In the following example, ELF displays a message if the title of the current window is not equal to the variable my_name.

```
WHILE CURRENT_WINDOW_TITLE@() <> my_name
    INFO_MESSAGE@("Sort ready, select the spreadsheet
            again")
WEND
```

Loop Branching within a WHILE Loop

ELF allows you to prematurely terminate an iteration of a WHILE loop or the entire WHILE loop.

The NEXT WHILE statements tells ELF to skip the remainder of the statements within the loop's body and resume execution at the top of the WHILE loop. The syntax for this statement is:

```
NEXT WHILE
```

The BREAK statement tells ELF to skip the remainder of the statements with the loop's body and resume execution with the statement following the WHILE loop. The syntax for this statement is:

```
BREAK WHILE
```

Here is an example that uses both of these statements

```
WHILE (TRUE)
    ...
    if COND = -1
        BREAK WHILE
    if COND = 1
        NEXT WHILE
    ...
WEND
```

The WHILE statements control the loop branching within the WHILE loop. If the COND variable is set to -1, the loop is terminated. If COND is set to 1, the statements following the IF statement down to the WEND statement are skipped. For any other value of COND, the statements after the IF statements are executed.

The BREAK WHILE and NEXT WHILE can only cause a branch to occur within the current loop. That is, even if you nest a WHILE loop

within a WHILE loop, you can only loop branch within the current WHILE loop. For example, you cannot break free of both loops with one BREAK statement.

XOR Operator

XOR is both a logical operator and a bitwise operator.

As a logical operator, XOR is used primarily in conditional expressions. For an expression containing an XOR operator to evaluate to TRUE, one of the objects in the expression must be TRUE, but not both.

XOR can also be used for bitwise operations. For example, the following statement displays the number 3 in an information box.

```
info_message@(5 XOR 6)  ' 0101 XOR 0110 = 0011=3
```

The following example contains an IF statement that tests x and y. If one of the variables is TRUE, but not both, the information message appears.

```
set test
var x, y
y = 1
if x XOR y
    info_message@("Either X or Y is true, but not both!")
endset
```

Refer to the sections "AND Operator" and "OR Operator" for related information.

2 Object-Oriented Concepts

Applix Builder uses object-oriented technology to implement applications. Object-oriented technology allows you to create an application with self-contained, reusable pieces of code.

A thorough knowledge of object-oriented programming is not necessary to begin programming Applix Builder applications. This chapter explains basic object-oriented concepts and how they are represented in Applix Builder.

This chapter covers the following topics:

- Object Oriented Concepts
- Objects in Applix Builder
- Object macros
- Summary of object classes

Object-Oriented Concepts

Applix Builder uses object-oriented technology to implement features. Object-oriented technology allows you to reuse or change code when creating an application. The following is an introductory explanation of object-oriented concepts, and how they relate to Applix Builder. Refer to books on object-oriented programming and fundamentals for more detailed information.

This section covers the following object-oriented concepts:

- Class
- Object
- Inheritance
- Method

The following sections in this chapter discuss the implementation of objects in Applix Builder. Refer to the Glossary in the *Applix Builder User's Guide* for definitions of terms used in this manual.

Class

A *class* is an abstract definition of the operations used to implement functionality. A class describes the expected behavior and attributes of a specific instantiation of the class. For example, you can look at the base classes for all Builder components by choosing View → Classes from the Browser window. A Class Browser dialog box appears containing the base classes, such as BaseClass, DialogBoxClass, and ButtonClass.

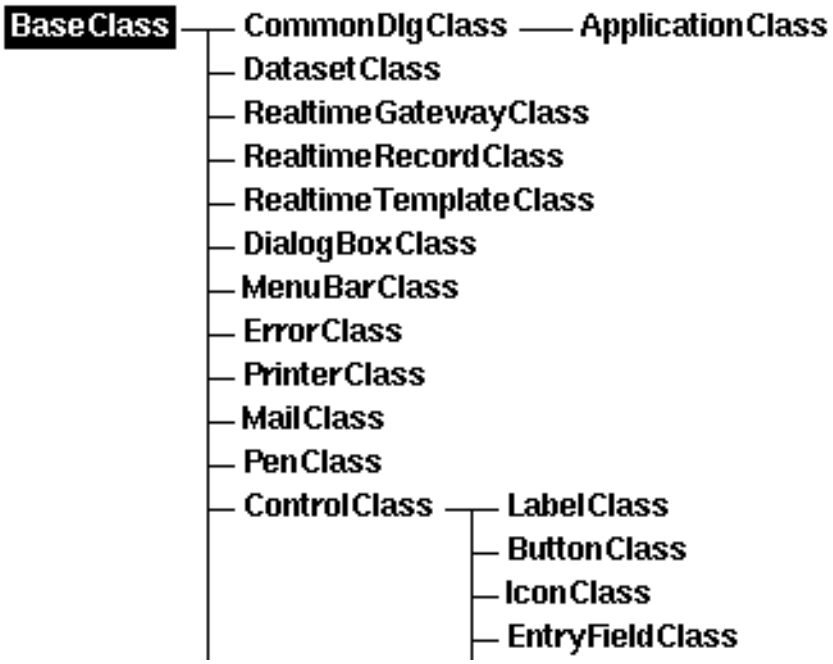


Figure 2-1 A Partial View of Base Classes

Object

An *object* is the concrete occurrence of a class. An object has the behavior and attributes of a class, as well as an identifiable state. The structure and behavior of similar objects is defined in their common class. The state of an object encompasses all of the object properties, plus the current values of the properties. An object property is usually static, while the current value of a property is usually dynamic.

Inheritance

When you create an object you can set or change the object inheritance. Object *inheritance* is the ability of an object to derive data attributes and methods from an existing class. A created object based upon an existing class has all methods and data members associated with the existing class, but is not limited to the existing object methods.

Inheritance also exists in classes. The partial view of base classes in the previous figure shows the class hierarchy. The ButtonClass has a set of defined methods, but it also inherits the methods in the ControlClass, the class it descends from. The ControlClass methods are implicitly defined in the ButtonClass, they are not actually listed in the available ButtonClass methods. The ButtonClass does not inherit methods from classes such as DialogBoxClass or ApplicationClass because it is not a direct descendant.

For example, select the ButtonClass in the Class Browser dialog box. When you choose the Class option in the Class Browser dialog box, the hierarchy of classes is apparent. The ButtonClass inherits the methods and attributes of ControlClass and BaseClass. All objects based on ButtonClass have this inheritance hierarchy.

Method

A *method* is an operation that can only access the data members and attributes of its own object. In Applix Builder, each base class defines a set of methods for manipulating its attributes. You can see the methods associated with an object in the Methods List area of the Class Browser window. Methods in Applix Builder consist of set (->) and get (<-) methods. The right arrow (->) and and left arrow (<-) visually identify a method in the On-line Help Search dialog box and help topics as a set or get method. Use set methods to set attributes and

information for an object. Use get methods to retrieve information from an object.

All Applix Builder base class methods end with the @ character. You cannot overwrite or replace a base class method ending with the @ character.

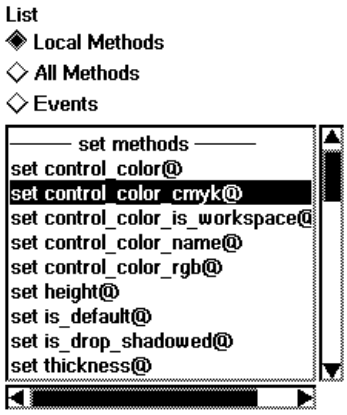
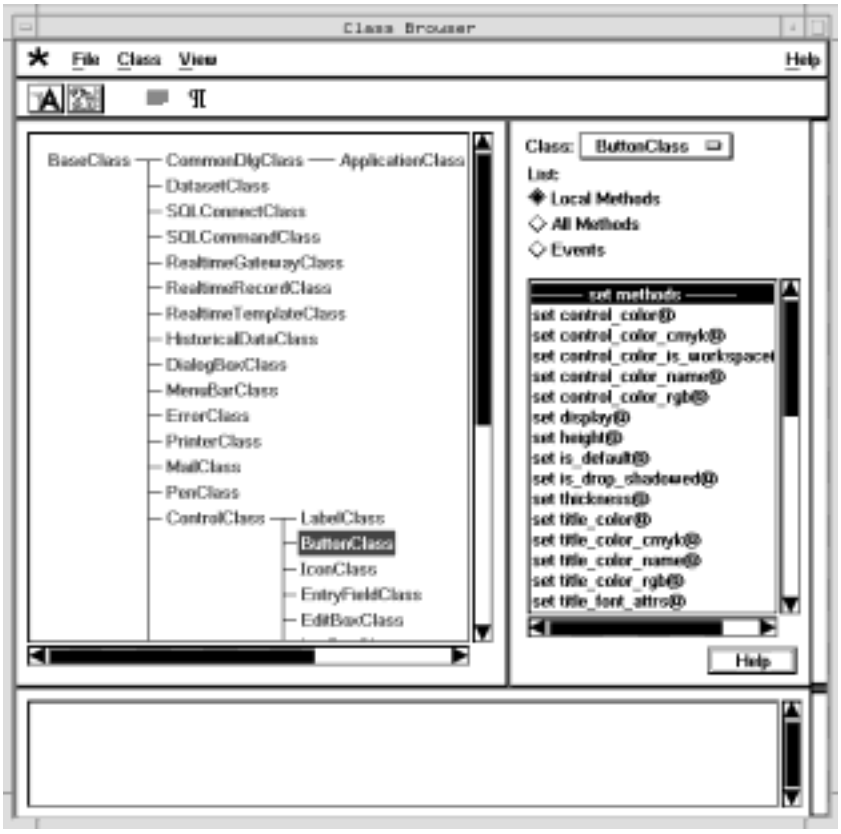


Figure 2-2 Object Methods in the Methods List area

You can edit the source of any object to add new methods to an object, add macros called by an object, or to modify or replace inherited classes. See Chapter 3, "Editing Object Methods", for information on editing an object source.

Objects in Applix Builder

Applix Builder creates applications with sets of objects. All objects in Applix Builder inherit attributes from a set of base classes. You can look at the base classes for all Builder components by choosing View → Classes from the Browser window. The Class Browser dialog box appears containing the base classes.



All base classes inherit their attributes from the BaseClass class. The BaseClass class contains all the methods and attributes defining objects in Applix Builder.

Builder Programming

Builder programming allows method name overloading. The same name can be used for set and get methods. The compiler/interpreter resolves which method to call with following rules:

- If a method is called as the right hand part of an expression, a get method is called, such as i = this.value@.
- If a method is called as a procedural call or is located on the left hand side of an assignment statement, a set method is called, such as this.value@ = 3.

The following are examples of set and get method calls. In the following example the open@ method is called as a procedural call, not expecting any return value, so a set method is called.

```
dbox.open@(arg1, arg2, arg3)
```

In this example the control_color@ method is called in an assignment statement on the left hand side, so a set method is called. The statements are equivalent, you can use either format to assign values with a set method.

```
thing.control_color@ = "red"
thing.control_color@("red")
```

In the following example the width@ method is called as part of an expression, so a get method is called. The get method value@ is also called.

```
new_value = dbox.width@
new_value = dbox.width@ + this.value@
if (this.value@ > 100) ...
```

Builder programming has requirements in addition to standard ELF programming.

- The ELF directive @@@ OBJECTS indicates that the object method source contains Builder code. The ELF directive @@@ OBJECTS must be the first non-commented line in the file to activate Builder mode.

- The following keywords are used in Builder programming, in addition to the standard ELF reserved words. See Chapter 1, "Language Elements", in this manual for a list of reserved words.

endget	private
endset	public
get	set
object	this
objvar	

See Chapter 3 "Editing Object Methods", for information on object method source components, editing an object method source, and defining get and set methods.

Private Methods

You may have object methods that are only used within the object and should not be available outside of the object. Use the private keyword to create private methods that can only be called within an object. For example, the private method paint would be defined in the object method source as:

```
private set paint
        /* paint method actions  */
        ....
endset
```

A private method can only be called from within the object method source using the implicit this pointer. You cannot reference the method in other object method sources. The keyword this is described in the "Object Relationships" section later in this chapter.

Variable Declarations Using object and objvar

The keywords object and objvar are essential to programming in Applix Builder. Although they appear similar, they are used in different circumstances.

The keyword object is a declaration of a data type. Use the object data type just as you would use ELF data types declared with the format keyword. A variable of type object is declared as:

```
var object dbox
```

The variable dbox in this example is of type object. You can use methods to access and set the variable's information.

The structure and state of properties are defined through object variables. The keyword objvar is used to declare local object data. Object data is entirely protected and bound to the object. Only object methods can reference these variables. Object data is declared as:

```
objvar object_width
objvar object_height
```

The keyword public makes objvar variables accessible outside of the object. Use a public objvar to set or get object data without methods. In the following example, the variable grape is defined as a public objvar.

```
public objvar grape
```

You can directly set or get the variable's data without calling an object method. For example, to set the public objvar grape to 5 use the following:

```
this.grape = 5
```

You can make an objvar public and have a set method with the same name. The set method overrides direct access to the objvar. This allows you to provide quick access for retrieving object information while verifying any information set in the object. For example, an object containing the public objvar age can also have a set method age to ensure a valid age is set.

```
public objvar age

set age(val)
  if (val < 0) or (val > 125)
    error@(1,"Not a valid age",val)
  age = val
endset
```

Object Relationships

When you write methods or macros for an object, reserved words are available to refer to objects and their related objects. The word this refers to the current object. For example, if an object has a method called display_contents, the object calls the method as follows:

```
this.display_contents
```

The words parent, child, and sibling refer to the relation of other objects to this. parent is the object this directly descends from. For example, in the following, where Button is this, Dialog1 is the parent of Button and Button.2, ChildButton is the child of Button, and Button.2 is a sibling of Button.

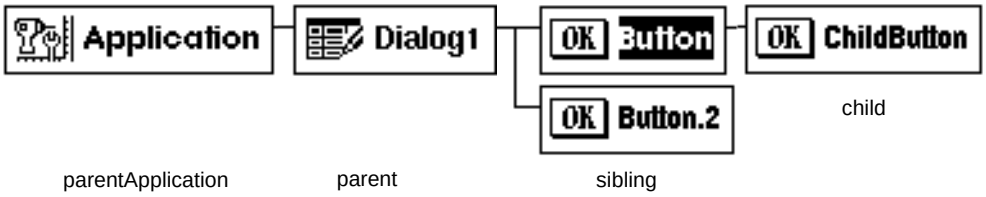


Figure 2-3 Relationship of Objects to Button Object

The parentApplication is the root object for any Builder application. The parentApplication object contains methods to control child objects in the application. The get method application@ returns the application object.

For example, given the Button object of the previous diagram, you could write a clicked_event in the Button source to dismiss the dialog box containing the button. The following is a sample clicked_event:

```
set clicked_event
  var object app
  app = this.application@
  app.dbox_close@(this.parent@)
endset
```

The event creates an object app containing the application of this (Button). The Button object does not contain the methods to close a dialog box. The application object app does contain a method to close dialog boxes, so app must invoke the method to close the dialog box of this.parent@ (Dialog1).

Object Events

Events are user-defined operations called by the Applix run-time environment when an action occurs. Each object has associated events. Applix Builder applications are event-driven; clicking on a button or typing in an entry area initiates a course of action. If the event

operations are not defined for an object, no object action occurs in response to the event. Each object has at least the following events:

error_event	Called for error with the object. This event should contain error handling code if you plan to trap errors that could be thrown by this object.
initialize_event	Called before an object is created. For dialog boxes and dialog box controls the event is called before a dialog box is displayed. This event should contain all initialization of objects before they are displayed or used in a dialog box.
terminate_event	Called before an object is destroyed. For dialog boxes and dialog box controls the event is called before a dialog box is closed. This event should contain all operations that "clean up" before a dialog box is closed, such as freeing memory, removing local variables, or resetting default text attributes.
time_out_event	Called when the time limit set in the object with the timer@ method expires. Use this event to perform timed actions in the application that are related to the object.

All objects in dialog boxes also have the following events:

resize_event	Called after dialog box is resized. This event should contain all object operations to reposition or resize objects after a dialog box is resized.
update_event	Called after dialog box is displayed. This event should contain all object operations to update object information, or set object information applicable after a dialog box appears.

Objects may have additional events to control actions, such as the clicked_event for ButtonClass objects. The sequence of initialize_event calls when you run an application is shown in the following figure.

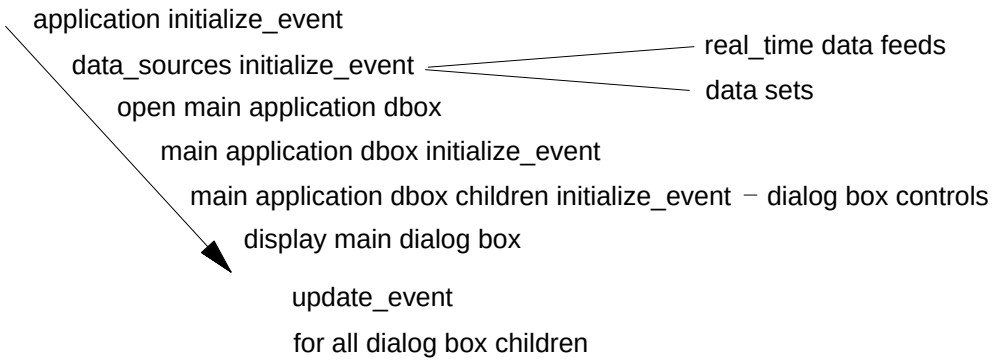


Figure 2-4 Sequence of Object initialize_event Calls

When you run the application:

1. The application initialize_event is called.
2. The initialize_event for each object is called, in the order that they appear in the Browser, except for dialog boxes. For each Real Time RealtimeGatewayClass object, the RealtimeGatewayClass object initialize_event is called, then the initialize_event for each of the children, the RealtimeRecordClass and RealtimeTemplateClass objects, are called.
3. The main application dialog box is opened.
4. The main application dialog box initialize_event is called.
5. The initialize_event for every dialog box control is called, in the order that they appear in the Browser.
6. The dialog box is displayed.

7. The update_event for each dialog box control is called after the dialog box is displayed.

When you open any other dialog boxes in an application, a similar sequence occurs. When a dbox_open@(*name*) method is called:

1. The dialog box initialize_event is called.
2. The initialize_event for every dialog box control is called, in the order that they appear in the Browser and Connector tools.
3. The dialog box is displayed.
4. The update_event for each dialog box control is called after the dialog box is displayed.

The sequence of events is reversed when exiting a dialog box or application. The terminate_event for each dialog box control is called, in the reverse of the order that they appear in the Browser window, then the dialog box terminate_event is called. After the main dialog box terminate_event, the terminate_event for each data source is called. The application terminate_event is the last event called before exiting the application.

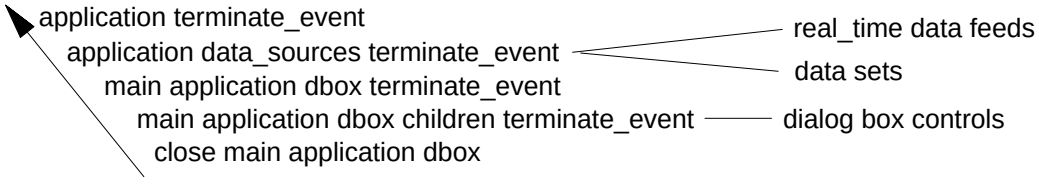


Figure 2-5 Sequence of Object terminate_event Calls

Object Notation

There are four forms of notation used in method calls:

.	method invocation
!	literal
*	broadcast
^	call inherited

Method Invocation (.)

The . notation indicates method invocation. For example, in the following example, the object app_object invokes the method dbox_open@ to open the dialog box Main_Dbox.

```
app_object.dbox_open@("Main_Dbox")
```

If app_object is the current object, the implicit this pointer can be used to call the method.

```
this.dbox_open@("Main_Dbox")
```

You can invoke multiple methods on one line, using the results returned by one method as the argument for the next method. For example, the following lines

```
var object app
app = this.application
app.dbox_open@("Main_Dbox")
```

can be replaced with

```
this.application@.dbox_open@("Main_Dbox")
```

The current object, referenced with this, gets a reference to the application object with the get method application@, then uses that reference to invoke the ApplicationClass set method dbox_open@.

Literal (!)

The ! notation indicates a literal string method invocation. For example, in the lines

```
event_name = "open@"
dbox.!event_name
```

The literal string open@ is assigned to the variable event_name. The dbox object uses the literal string to invoke the open@ method.

Broadcast (*)

The * notation indicates a broadcast of a method. You can broadcast to a single object, or to a group of objects. You can broadcast a set or get method to a single object, while you can broadcast a set method only to a group of objects. You cannot broadcast a get to a group of objects because get methods return values, and there is no way to assign the returned values of a group of objects in one statement. The variables receiving a broadcast must be declared as object variables in the method, using the following definition:

```
var object sample, multi_sample
```

You can declare multiple object variables on the same line. You can use the broadcast of a method on a single object if you do not want an error thrown on failure. For standard invocation of methods, if the method does not exist, then an error is thrown. For example, for the single object sample, the broadcast of sample, as in the following line, attempts to invoke the method dbox_close@.

```
sample.*dbox_close@
```

If the method dbox_close@ exists, it is invoked, and application execution continues. If the method does not exist, no action is taken, no error is thrown, and the application execution continues.

For a group of objects, such as the children of a dialog box, the broadcast invokes a set method for all objects, if the method exists, instead of invocation on an individual basis. For example, the object multi_sample references a set of objects, the children of a dialog box. The following broadcast hides all the children of the dialog box and continues application execution.

```
multi_sample.*is_hidden@(TRUE)
```

If the method does not exist, no action is taken, no error is thrown, and the application execution continues.

Use broadcast on references to a set of objects because you cannot de-reference the set as an array. For example, you cannot reference individual elements of multi_sample using an array reference, such as multi_sample[i]. A broadcast is also a simpler and more efficient way of invoking methods in an array of objects, instead of writing a loop to reference each array item.

Call Inherited (^)

The ^ notation indicates a direct call of an inherited method. Use the call inherited invocation if an object has a method that overrides an inherited method. For example, if you create an object first with a method called paint, inheriting from a class that also has a method paint, the invocation by the first object

```
this.paint
```

invokes the paint method in first, not the paint method defined in the class. The invocation by the first object

```
this.^paint
```

invokes the paint method defined in the class, the method the first object would have inherited if the method was not overridden.

You can only use the call inherited method from within the local method of the same name. The following is an example of a method M that overrides the inherited method M.

```
set M(arg1)
  this.^M(arg1)   'Call the M this would inherit
endset
```

The call inherited invocation is used primarily in object initialization and termination. For example, during object termination, an inherited object destroys its own information, then destroys the information it inherits. The inherited object should have its destroy method call the destroy method it inherits.

Call inherited invocations can also be broadcast and literal invocations. The following are different ways of calling an inherited method.

Direct	obj.^method
Broadcast	obj.*^method
Literal	obj.^!name
Broadcast literal	obj.*^!name

Object Macros

Use the Applix Builder ELF macros to program general purpose or application-specific actions. The following is a list of Applix Builder ELF macros and a summary of their use.

BUILDER_APPLICATION@

Runs an Applix Builder application in a new task.

BUILDER_APPLICATION_DLG@

Runs Applix Builder in a new task.

BUILDER_APPLICATIONS_IN_MEMORY@

Returns a list of ApplicationClass objects in memory matching the passed name.

BUILDER_INSTALL_DISTRIBUTION@

Installs an Applix Builder distribution file with the the Install Builder Distribution dialog box.

BUILDER_READ_DOC@

Reads an Applix Builder application into a variable of builder_docinfo@ format.

BUILDER_WRITE_DOC@

Writes an Applix Builder application to file.

BUILDER_WRITE_TURBO_DOC@

Writes an Applix Builder application to file in turbo format.

IS_OBJECT@

Indicates if passed variable is an object.

OBJECT_CREATE@

Creates an object.

OBJECT_DESTROY@

Removes an object and all children.

OBJECT_EXISTS@

Returns object existence.

REMOTE_OBJECT_CREATE@

Creates a remote object.

REMOTE_OBJECT_DESTROY@

Removes a remote object and all its children.

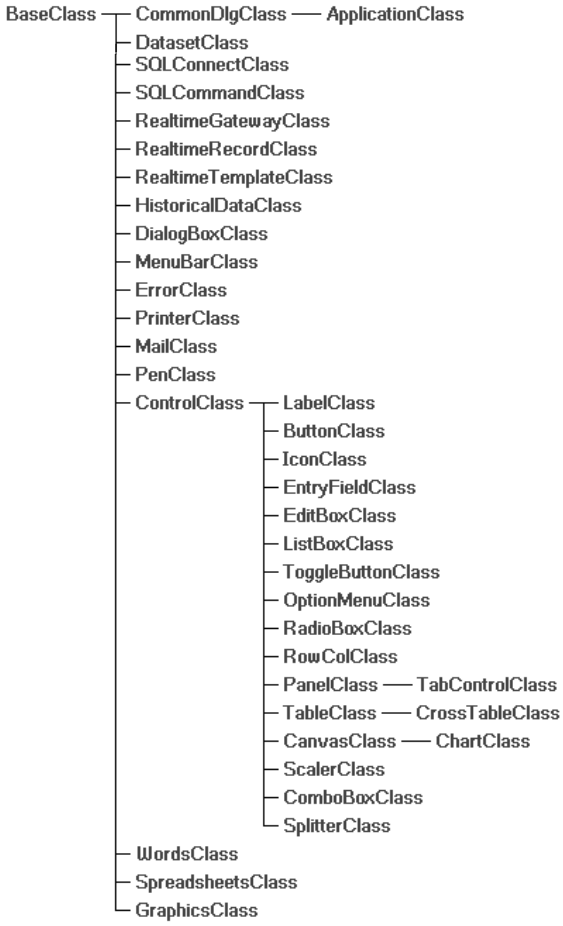
REMOTE_OBJECT_FIND@

Returns a handle to a remote object.

Summary of Object Classes

A class describes the expected behavior and attributes of objects that inherit attributes from the class. Applix Builder base classes contain the methods you need to perform application operations. You can create classes, as described in Chapter 2 of the *Applix Builder User's Guide*, "Using the Browser Tool", to add functionality beyond that supplied by the base classes.

The following is a diagram of classes as they appear in the Class Browser window, and an alphabetical summary of each class. See "Using the Browser Tool" for information on using the Class Browser window.


BaseClass
CommonDlgClass
ApplicationClass
DatasetClass
SQLConnectClass
SQLCommandClass
RealtimeGatewayClass
RealtimeRecordClass
RealtimeTemplateClass
HistoricalDataClass
DialogBoxClass
MenuBarClass
ErrorClass
PrinterClass
MailClass
PenClass
ControlClass
LabelClass
ButtonClass
IconClass
EntryFieldClass
EditBoxClass
ListBoxClass
ToggleButtonClass
OptionMenuClass
RadioBoxClass
RowColClass
PanelClass
TabControlClass
TableClass
CrossTableClass
CanvasClass
ChartClass
ScalerClass
ComboBoxClass
SplitterClass
WordsClass
SpreadsheetsClass
GraphicsClass

ApplicationClass	Use the ApplicationClass methods to control the actions and contents of an application. The ApplicationClass object is the top-level object in an application. An application can contain only one ApplicationClass object.
BaseClass	The BaseClass contains methods used by all objects in Applix Builder. Use these methods from any of the Builder classes to perform basic actions.
ButtonClass	The ButtonClass contains methods used by dialog box push buttons. Use a ButtonClass object to perform an immediate action, such as opening or closing a dialog box.
CanvasClass	The CanvasClass contains methods used to control a dialog box canvas. Use a canvas to display text and images. Use PenClass methods in conjunction with CanvasClass methods to render text and images on a CanvasClass object.
ChartClass	The ChartClass contains methods used to create a chart in a dialog box.
ComboBoxClass	The ComboBoxClass contains methods used by the combo box control. Use a combo box in a dialog box to provide a multiple item display in a limited area.

CommonDlgClass	The CommonDlgClass contains the methods used to access the common application functions and dialog boxes. See Chapter 11, "Advanced Topics", for a list of common dialog box methods.
ControlClass	The ControlClass contains the methods and events used by all dialog box control objects. The ControlClass is an abstract class, do not create objects based directly on the ControlClass.
CrossTableClass	The CrossTableClass contains methods used to control a dialog box cross table. The CrossTableClass inherits attributes from the TableClass. A cross table summarizes the data from two columns in a data set and displays it in a grid format. You can choose a row, column, and type of value for a cross table. A value uses an aggregate function, such as count(*) or avg(), to summarize cross table information.
DatasetClass	The DatasetClass contains methods used for database access and information manipulation. Use DatasetClass methods to perform database operations. You can use the methods to enhance a data set created with the Source tool, or to programmatically retrieve database information.

DialogBoxClass	The DialogBoxClass contains methods used to control the appearance and attributes of a dialog box in the application.
EditBoxClass	The EditBoxClass contains methods used by dialog box edit boxes. An edit box displays a large amount of text that you can manipulate. You can add or remove information in an edit box. The text in an edit box automatically wraps, so the text will adjust to any edits. A vertical scrollbar allows you to move to different parts of the text in the edit box.
EntryFieldClass	The EntryFieldClass contains methods used by dialog box entry fields. An entry field is a text entry area where the user can type information. Use an entry field to get string information for use in an application.
ErrorClass	The ErrorClass contains methods for writing error handlers. If you do not catch errors, the Applix Builder run time environment handles the errors. Write an error_event to handle the error object created by the application when an error is thrown. If an error_event does not exist, the error is posted in a message box, then the application is terminated.

GraphicsClass	The GraphicsClass contains methods for programming the Applixware Graphics application. The methods are the equivalents of the GR_ ELF macros.
HistoricalDataClass	The HistoricalDataClass contains methods for retrieving historical information through data feeds.
IconClass	The IconClass contains methods used by dialog box bitmaps. Use an IconClass object to place a decorative image in a dialog box.
LabelClass	The LabelClass contains methods used by dialog box labels. Use a label to define controls and areas in a dialog box.
ListBoxClass	The ListBoxClass contains methods used by dialog box list boxes. A list box is a group of items displayed inside a box. You can use vertical and horizontal scroll bars to scroll through entries if necessary. Some list boxes allow more than one item to be chosen at a time, while other list boxes allow only one item to be chosen at a time.
MailClass	The MailClass contains methods for creating and sending e-mail from an application.

MenuBarClass	The MenuBarClass contains methods used by dialog box menu bars. Use MenuBarClass methods to define or control menu bar actions in a dialog box.
OptionMenuClass	The OptionMenuClass contains methods used by dialog box option menus. An option menu offers a menu of options from which to choose, with only the current choice visible in the dialog box.
PanelClass	The PanelClass contains methods used by dialog box panels. Use a panel to separate groups of controls in the dialog box.
PenClass	The PenClass contains methods used to set pen attributes. Use PenClass methods in conjunction with CanvasClass methods to render text and images on a CanvasClass object. Refer to XWindows information on graphics context (gc) for an understanding of how to use a PenClass object.
PrinterClass	The PrinterClass contains methods used to print text, images, and charts. Use PrinterClass objects to print the information contained in an application to PCL5 or PostScript printers.

RadioBoxClass	The RadioBoxClass contains methods used by dialog box radio button groups. A radio button group is a group of items from which only a single item can be chosen at a time. Normally, radio button groups are a collection of logically associated items.
RealtimeGatewayClass	The RealtimeGatewayClass contains methods used to define a Real Time gateway in an application.
RealtimeRecordClass	The RealtimeRecordClass contains methods used to define the information records ccontained in a Realtime GatewayClass object.
RealtimeTemplateClass	The RealtimeTemplateClass contains methods used to define a set of common field information to retrieve across a group of records.
RowColClass	The RowColClass contains methods used by dialog box RowCol widgets. A RowCol widget is similar to an option menu. Use a RowCol widget to display either text or bitmaps in an expandabale, multiple column menu. An option menu is limited to a single column of text.

ScalerClass	The ScalerClass contains methods used by dialog box scales. Use a scale to gradually increment and decrement values. A scale consists of a slider and sliding area. A scale can be used alone, or with an entry area to provide exact feedback on values. A scale is only used horizontally.
SplitterClass	The SplitterClass contains methods used by dialog box splitters. A splitter is a diamond-shaped marker. Use a splitter to resize controls in a dialog box. A splitter moves along a vertical area within a dialog box.
SpreadsheetsClass	The SpreadsheetsClass contains methods for programming the Applixware Spreadsheets application. The methods are the equivalents of the SS_ ELF macros.
SQLCommandClass	The SQLCommandClass contains methods to programmatically query and update a database. Use this class with SQLConnectClass to manipulate database information.
SQLConnectClass	The SQLConnectClass contains methods to programmatically connect to a database. Use this class with SQLCommandClass to manipulate database information.

TabControlClass	The TabControlClass contains methods for using a tab in a dialog box. Use a tab to create a multiple-layered dialog box.
TableClass	The TableClass contains methods used by dialog box tables. A table displays information in columns and rows. A table can have horizontal and vertical scroll bars so that you can move through the table information.
ToggleButtonClass	The ToggleButtonClass contains methods used by dialog box toggle buttons. A toggle button can either be turned on or turned off. Use a toggle button control when you have a single item that can either be chosen or not chosen.
WordsClass	The WordsClass contains methods for programming the Applixware Words application. The methods are the equivalents of the WP_ ELF macros.

3 Editing Object Methods

The Applix Builder Source and Designer tools allow you to create a basic application. To make a robust, full-featured application requires programming with methods. This chapter covers the following topics:

- Using methods
- Editing an object method source
- Using the Edit Source window

Using Methods

Methods set the information and attributes for an object. All objects in Builder inherit methods and attributes from a class. Inheriting methods means you do not need to redefine a set of methods for each object you create. However, you may want to write your own methods to implement features for a specialized object. You also need to define events for object actions.

Applix Builder applications are event-driven; clicking on a button or typing in an entry area initiates a course of action. If the event operations are not defined for an object, no object action occurs in response to the event.

For example, an object that inherits from the ButtonClass has, by default, all the methods to set or get a button object's attributes. A button object also has a clicked_event that is called by the application when the button is pressed. You must use methods and macros to define the clicked_event actions to perform a task, such as dismissing or opening a dialog box.

You can define methods for an object in the object's method source. An object method source contains all methods, macros, and events specific to an object.

An object's methods or object functions are completely public. An object or macro calls methods through its object handle. An object can have associated data, but the data is entirely private to the object. Only an object's methods can read or write to its data. See Chapter 2, "Object-Oriented Concepts", for a further description of objects, classes, and methods.

The following sections in this chapter illustrate how to use the Edit Source window to make changes to an object method source.

Editing an Object Method Source

You can, as stated previously, write your own methods to implement features for a specialized object, or to define events for object actions. This section demonstrates how to edit an object method source.

Opening an Edit Source Window

You can open an Edit Source window from the Browser window or Connector dialog box.

To open an Edit Source window from the Browser:

1. Click on an object in the Browser area.
2. Click on the **Source** icon, or choose **Object → Edit Methods**.

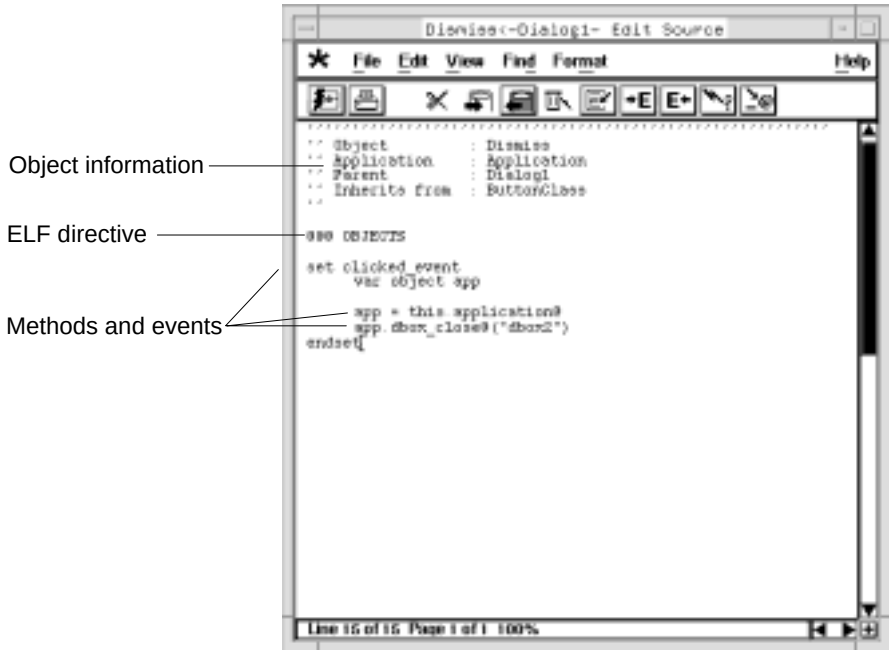

Source icon

To open an Edit Source window from the Connector:

1. Choose a dialog box with the Dialog Box option.
2. Click on an object in the Dialog Box list area
3. Click on **Source**.

Source Components

When you open an object method source in an Edit Source window, the object method source contains the following components:



Object information — The object information is a set of commented lines indicating the object name, application name, parent object, and inheritance class.

ELF directive — The ELF directive @@@ OBJECTS indicates the object method source contains Builder code. The ELF directive @@@ OBJECTS must be the first non-commented line in the file.

Methods, events — The methods and events that are unique to the object.

The object source in the previous diagram is for an object named Dismiss inheriting attributes from the ButtonClass. The clicked_event was inserted in the object method source. When you create an object

the only information in the object method source is the object information and the @@@OBJECTS ELF directive.

The first line of a method or an event contains the type of method or event and the method or event name. For example, the first line of the clicked_event is:

```
set clicked_event
```

The last line of a method or event is endset or endget, depending on whether a method is, respectively, a set or get method. Events can also end with endset or endget. If an event returns a value, it should end with endget. The body of a method or event contains method and macro calls to set attributes or perform actions.

Method Names and Arguments

The name of a method must be unique within the object, and must adhere to the following naming conventions:

- Names must begin with a letter of the alphabet.
- Names can contain alphabetic, numeric, or underscore (_) characters.
- Names cannot include spaces.

Method names must not include:

- the "#" character.
- The @ symbol, $ symbol, or _ (underscore) symbol as the last character in the macro name. By convention, only ELF built-in methods and macros can use these symbols as the last character in the method or macro name.

A method can receive one or more arguments, if required. If more than one argument is passed to the method, the arguments are separated by commas:

```
set Example(arg1, arg2)
```

If a method does not require arguments, then only the set or get keyword and the method name are necessary to define the beginning of the method.

Different objects in an application can have methods with the same name. This is useful for broadcasting a method call to a group of objects. To avoid confusion in your application, however, you may want to use unique method names for each object within your application. See Chpater 2, "Object-Oriented Concepts", for information about using a broadcast in your application.

Defining a set Method

Use set methods to set attributes and information for an object. A set method defines the interface that other objects use to set the object information or actions.

A set method is defined with the the set and endset keywords. The set method consists of all the lines of code between these keyords. The name of the set method follows the set keyword on the same line. For example, the following set method, color_and_font, sets the title color of the dialog box control by name to Red and the title font to Helvetica.

```
set color_and_font
        this.title_color_name@("Red")
        this.title_font_name@("Helvetica")
endset
```

Defining a get Method

Use get methods to retrieve information from an object. A get method defines the interface that other objects use to retrieve information from the object.

A get method is defined with the the get and endget keywords. The get method consists of all the lines of code between these keyords. The name of the get method follows the get keyword on the same line. A get method must return information, so there must be at least one return statement in the body of the get method. For example, the following get method, was_it_on, returns a string representing the is_on@ state of a ToggleButtonClass object.

```
get was_it_on
        IF this.is_on@
                return("Toggle On")
        ELSE
                return("Toggle Off")
endget
```

A get method can return any supported ELF data type. For complex data types, the variable receiving the results of the get method should be defined as the same type of the return data.

Defining an Event

An event is called by the application for an object action. In the case of a ButtonClass object, the clicked_event is called by the application when a button is pressed in a dialog box. An event, like a get or set method, is unique to an object method source.

For example, a dialog box contains two ButtonClass objects, Dismiss and Open. Dismiss and Open both contain a clicked_event, but they perform different actions.

The clicked_event for the Dismiss object is defined as:

```
set clicked_event
        var object app
        app = this.application
        app.dbox_close@(this.parent@)
endset
```

The Dismiss object closes the dialog box. The event gets a handle to the parent application object and uses the dbox_close@ method to close the dialog box containing the button. The dbox_close@ method is not available in a ButtonClass object, only from an ApplicationClass object. The get method parent@ returns the handle of the Dismiss (this) object's parent, the dialog box containing the button. The handle is used as an argument for the dbox_close@ method.

The clicked_event for the Open object is defined as:

```
set clicked_event
        var object app
        app = this.application@
        app.dbox_open@("dbox2")
endset
```

The Open object opens another dialog box in the application. The event gets a handle to the parent application and uses the dbox_open@ method to open a dialog box. The name of the dialog box to open is passed as an argument to the dbox_open@ method.

An initialize_event is called to initialize an object before it used in an application. You can use the initialize_event to get refrences to other objects in the application once during the application execution, instead of repeatedly getting references during application execution.

This can enhance the performance of your application, as well as improve the efficiency of memory used by your application. For example, in the following object method source, the initialize_event gets a reference to an object that is used every time in the clicked_event.

```
@@@ OBJECTS
objvar object sibling
set initialize_event
        ' Get the reference to the toggle button
        sibling = this.sibling@("Toggle Button")
endset
set clicked_event
        ' Turn the toggle button off
        sibling.is_on@ = FALSE
endset
```

See Chapter 2, "Object-Oriented Concepts", for information about objects, methods, and relationships of objects in an application.

Relative Path Names of include Files

Relative path names for include files can be used in object method sources. An include defined as

```
include "foo.h"
```

is searched for in the current directory. If the file is not found in the current directory, the file is searched for in the directories in your ELF search path, as defined in the ELF_SEARCH_LIST@.

An include defined as

include "./foo.h"

is searched for in the current directory only.

An include defined as

include "/*directory*/foo.h"

is searched for in the *directory* path only.

Using the Edit Source Window

You can start an Edit Source window from the Browser or Connector tools. The Edit Source window provides the necessary features for editing an object method source. Use the Edit Source window to quickly edit, print, compile, or format the object method source.

Viewing Options

The Edit Source window offers various viewing options:

View → ExpressLine	Turn on to display a bar of buttons for frequently used operations.
View → Ruler	Turn on to display the graphical ruler.
View → Boundaries	Turn on to display all formatting boundaries in the document such as headers, footers and margins.

View → Delete Error Messages	Turn on to remove the error messages inserted into the source as a result of compiling.
View → Zoom	Choose one of the magnification size options to display the document in a variety of magnification sizes, larger or smaller. Sizes less than 100% will shrink the display of the text; sizes greater than 100% magnify the text. 100% displays the document at full size. The Zoom size does not affect the printed document. The document always prints at 100% size.

Inserting Text

You can type text into a source document after opening it. Text automatically word-wraps.

Inserting Special Characters

You can include special characters in your object method source. Choose Format → Special Characters and select the group and character you want to insert.

NOTE: If you save your source document in ASCII format, Symbol and Dingbat fonts will be lost.

Character and Paragraph Settings

You can set the typeface, size, bold, and italic text attributes in object method sources using the Format → Character Settings option. Select the text or line to change and choose the appropriate typestyle from

this dialog box. If nothing is selected when you change a setting, the default attributes are changed throughout the document.

You can set the paragraph spacing, indent levels, hyphenation and justification using the Format → Paragraph Settings option.

NOTE: If you save your object method source in ASCII format, all paragraph and character settings will be lost.

Promoting and Demoting Text

The Edit Source window offers the ability to Promote and Demote lines of text using Format → Promote and Format → Demote. This is useful for differentiating nesting levels when writing in an object method source.

NOTE: If you save your object method source in ASCII format, indent level information will be lost.

Shift Case

To change the case of selected text, choose Edit → Shift Case. Choose this option again if needed to change to the case you want.

The first word in the selected text determines the initial case setting. The case changes in the following order:

UPPER CASE
Initial Capitals
lower case

That is, if you select UPPER CASE text and then choose Edit → Shift Case, the text changes to Initial Capitals. Choose Edit → Shift Case

again to change the text to lower case. The next time you choose Edit → Shift Case, the text cycles back to UPPER CASE, and so on.

If the selected text is lower case to begin with, the first change is to UPPER CASE.

Mixed case, such as ApPLiX, changes to lower case (applix). The original mixed case cannot be retrieved using Edit → Shift Case. Choose Edit → Undo to return to mixed case.

Cutting, Copying and Pasting Text

Choose Edit → Cut to remove selected text.

Choose Edit → Copy to make a copy of the selected text to be pasted elsewhere in this or another document.

When you choose Edit → Cut or Edit → Copy, the selected text is placed in the clipboard. In the case of an Edit → Cut the text is removed from the object method source.

Once material is on the clipboard, you can use Edit → Paste to move the cut material or to insert the copied material into the object method source at the current cursor position.

NOTE: The contents of the clipboard are overwritten each time you use Edit → Cut or Edit → Copy.

Selecting All Text in an object method source

Choose Edit → Select All to select all text in the object method source.

Deleting Selected Text

Choose Edit → Delete to delete all currently selected text.

Unlike Edit → Cut, Edit → Delete does not save the deleted text in the clipboard. You cannot retrieve deleted text unless you use Undo.

Inserting a File

Choose File → Insert File to insert the contents of an ASCII file into the current Edit Source window. The Insert File dialog box appears, with a list of all files in the current directory. Choose a file from the current directory, or use the dialog box options to search for files in different directories. The file is inserted at the text cursor location in the source.

Undoing Changes

Choose Edit → Undo or press the UNDO key (F2) to restore your object method source to its condition prior to the last edit performed.

An edit is considered any operation that alters the contents of the object method source. For example, if you position the cursor at an insertion point and type a few characters, but then decide you want to delete the characters, you can choose Edit → Undo to restore the line. The Undo menu option changes dynamically to reflect the last action you performed, such as Undo Cut. This is the action that will be undone.

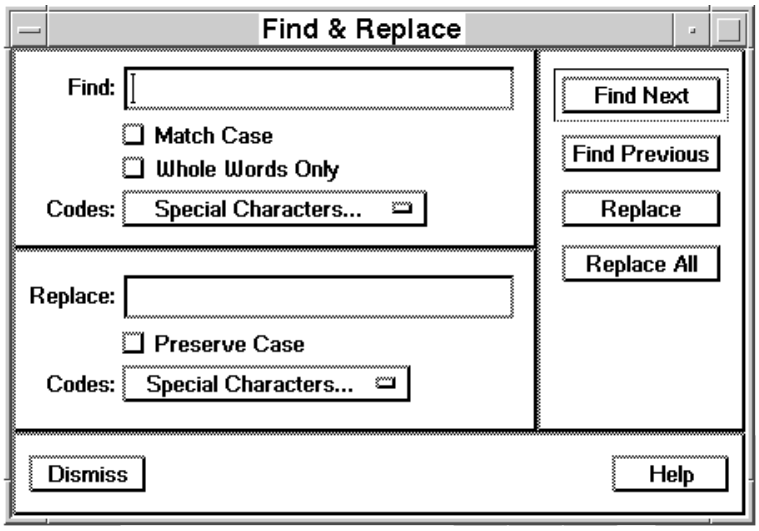
You can use Undo repeatedly to walk back through your actions up to the last File → Save operation. Redo restores any action you ''undid.''

You can use Edit → Redo or press the REDO key (SHIFT-F2) to redo an action you have just reversed using Edit → Undo. For example, if you delete material, then choose Edit → Undo to restore it, you can choose Edit → Redo to delete it again.

Searching For Text

To find and select text in a document:

1. Choose **Find → Find & Replace** to display the Find & Replace dialog box.



2. Type the text you want to find in the Find entry area.

   You can enter 1 to 80 characters including numbers, letters, symbols, blank spaces, and wildcards.

   Click on one of the Codes: options to search for a special character, the clipboard content, a word start or end, or a paragraph start or end.

3. Click on **Match Case** to find only text typed in the same case as the text you entered in the Find entry area.

Click on **Whole Words Only** to find only whole words including trailing spaces or punctuation marks.

4. Click on **Find Next** to initiate the search in a forward direction. Macros searches from the text cursor to the end of the document. After it reaches the end of the document, it then searches through headers, footers and footnotes.
5. Click on **Find Previous** to initiate the search in a backward direction. Macros searches from the text cursor to the beginning of the document.

Replacing Text

Choose Find → Find & Replace to substitute one text pattern with another. To replace text:

1. Complete the upper portion of the dialog box just as you would for searching, as explained in the previous section.
2. Type the text to substitute in the Replace entry area.
3. Click on **Preserve Case** to enforce the case in the Find box on the replaced text.
4. Click on one of the Codes: options to replace with a special character, macro, or the clipboard content.

After typing the search information, determine if you want to replace just the currently selected text or all occurrences of the search text.

Replace	Click here to replace just the current selection.
Replace All	Click here to replace all occurrences of the selected
Find Next	Click here to move forward to the next occurrence.

Find Previous	Click here to move backwards to the previous occurrence.
Cancel	Click here to ignore the find and replace information and return to the document.

You can also use the Find → Next and Find → Previous menu options to find subsequent occurrences of a word.

Moving Around in Object Method sources

Use other Find menu options to move the cursor to specific positions within the object method source. Use Find → Page to go to a specific page number.

Moving to the Next or Previous Error

After you have written a new macro or edited an existing one, you must use File → Compile & Save or File → Compile to check the syntax of your macros. If any syntax errors are found when the source is compiled, error messages are inserted in the document. Error messages are displayed with four asterisks (****).

Choose Find → Next Error to move the cursor to the next error message in the source.

Use Find → Previous Error to move backward to the previous error message.

To remove the error messages from your object method source document choose View → Delete Error Messages.

When no more errors are found with either Find → Next Error or Find → Previous Error, the cursor remains at the last found error and the terminal beeps.

Deleting Syntax Error Messages

To remove all syntax error messages generated as a result of choosing File → Compile & Save or File → Compile, choose View → Delete Error Messages.

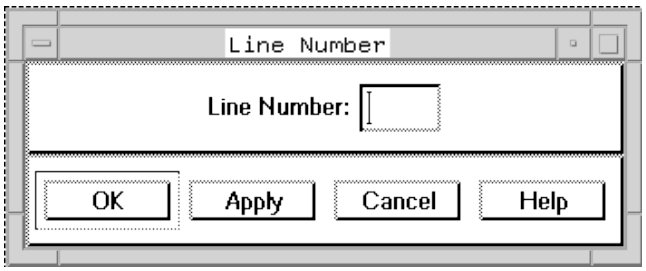
Error messages are also automatically removed when you print a document.

Moving to a Specific Line Number

The term *line* in this manual means all text in front of a paragraph marker.

To move the cursor to a specific line (paragraph) in a document:

1. Choose **Find → Line**.



2. Type the number of the line to which you want to move the cursor. Lines are numbered from 1.

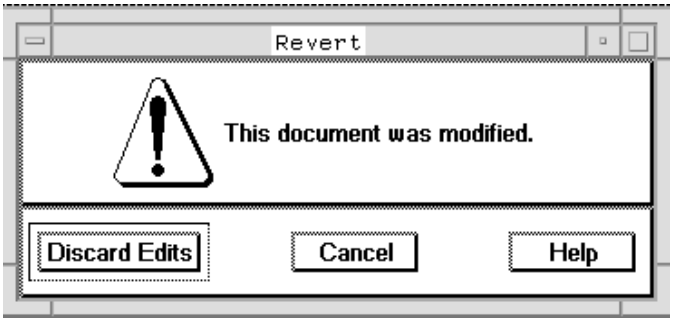
Reverting an Object Method Source

Choose File → Revert to eliminate all changes made to an object method source since the last time it was saved.

This option is a useful way to quickly eliminate changes if you find that you do not want to save any of the changes you have made.

To revert a document to its last saved state:

1. Choose **File → Revert**.



2. To revert the document, click on **Discard Edits**.

   To cancel the revert, click on **Cancel**.

Compiling an Object Method Source

Choose File → Compile to compile the methods and macros in the current Edit Source window. If a compilation error occurs, the errors are inserted into the file in a different typeface beneath the line of text where the error occurred.

Saving an Object Method Source

Choose File → Save to save the methods and macros in the current Edit Source window. You can save without compiling, so that you can save the object method source while it is an intermediate state.

Choose File → Compile & Save to compile and save the methods and macros in the current Edit Source window in one step.

If a compilation error occurs, the errors are inserted into the file in a different typeface beneath the line of text where the error occurred. Note that if you save an object method source in ASCII format, no formatting or special characters will be saved. If you subsequently revert, all visible formatting will be lost.

Saving an Object Method Source as an External File

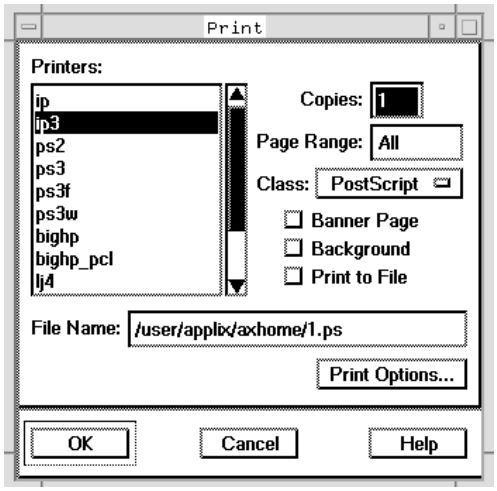
Choose File → Write File to save the object method source as an external ASCII file. When you choose this option the Save As dialog box appears. Use the dialog box options to set the file name, directory, and permissions. You can save a object method source to reuse pieces of code instead of rewriting code for similar methods.

Page Layout

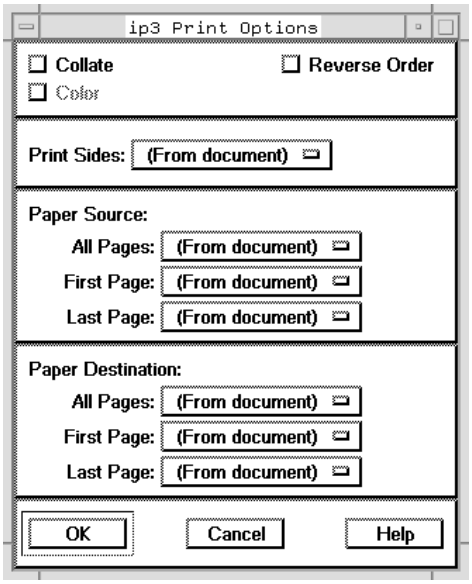
You can change the margins or print setup in the Edit Source window using Format → Page Setup. You can print in portrait or landscape orientation. Also, you can have headers and footers in object method sources using Format → Header & Footer.

Printing an Object Method Source

Choose File → Print to print a source. When you print a source with error messages, the error messages are removed. The selections in the Print dialog box reflect the choices made the last time you printed.



Click on the Print Options button to set the print options for the particular printer.



Using an External File Editor

You can edit files in an external editor by setting the External Editor Command in the Builder Preferences. Choose Edit → External Editor to launch an instance of the editor from the Edit Source window.

Exiting the Edit Source Window

To exit the Edit Source window, choose File → Exit.

If you have not made any changes to the current source, the object method source is closed and the Edit Source window is exited. If you have made a change to the current source, the Exit dialog box displays. Choose the appropriate button from the Exit box:

- Choose **Save** to save and compile the current source before it is closed.

- Choose **Discard** to close the current source without saving it. Any changes you made to the source since the last time it was saved are lost.
- Choose **Cancel** to cancel the File → Exit option.

4 Building Introductory Sample Applications

Using Applix Builder, you can construct a simple application in a few steps without programming. This chapter gives you step-by-step instructions on creating sample applications. This chapter covers the following topics:

- Building an introductory sample application
- Building an advanced application with two dialog boxes

Overview, Goals, Final Results

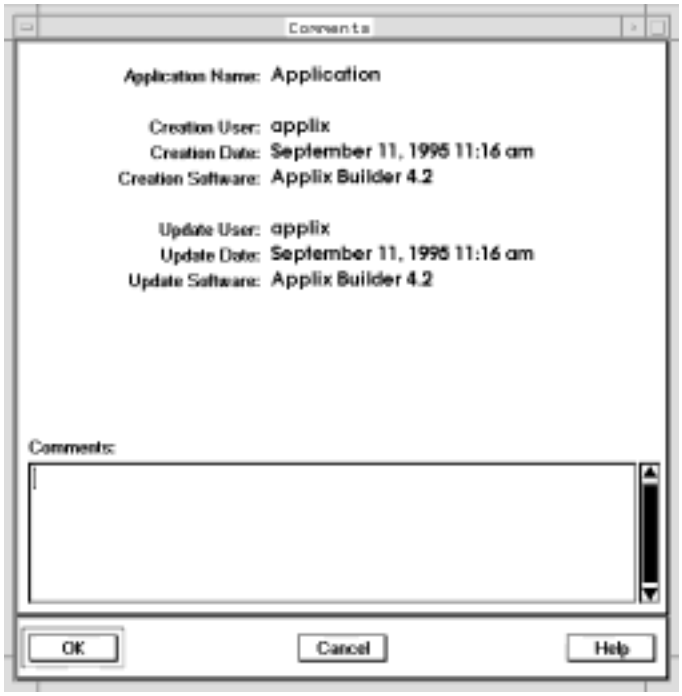
You can create quick, customizable applications using Applix Builder. This chapter focuses on the steps needed to create an introductory application See Chapter 24, "Building Advanced Sample Applications", to learn about building complex applications that access database or Real Time information.

When you complete this chapter you will have the basic steps for creating an application. The sample applications created in this chapter are complete, working products.

Commenting an Application

An application may go through multiple revisions before becoming a deployable application. Use the Comments dialog box to save information and comments about the application. Type all your information, such as creation, edit reasons, or purpose, in the Comments area.

Choose View → Comments to add comments to an application.



Previous comments appear in the Comments area. Type your comments in the area. Added comments are saved when you save the application.

Building an Introductory Sample Application

This section shows how to build an introductory sample application. The application consists of a single dialog box. The dialog box toggles the display of a label in the dialog box. This section covers:

- Creating a dialog box
- Editing object sources
- Running the application
- Saving and exiting the application

Creating a Dialog Box

Create a dialog box with the Designer tool. Every application requires at least one dialog box. This sample application contains one dialog box. To create a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type **Intro** in the Dialog Boxes entry area.
3. Click on **Create**.
4. Choose **Main** in the Type option.

5. Click on **Edit**. The designer window appears.

6. Choose **Controls → Label** and click in the designer window.

   A label appears in the designer window. Drag the label to the desired position in the dialog box.

7. Choose **Controls → Toggle Button** and click in the designer window.

   A toggle button appears in the designer window. Drag the toggle button to the desired position in the dialog box.

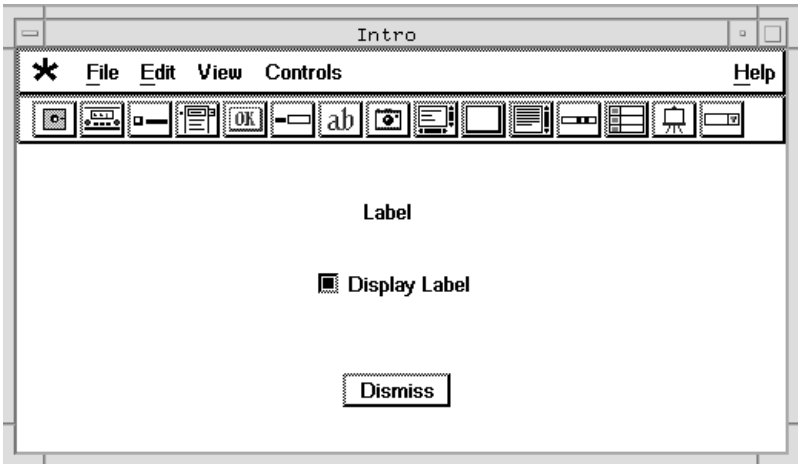
8. Double-click on the toggle button to change the toggle buttons attributes. The Toggle Button Attributes dialog box appears.

9. Double-click in the Title entry area.

10. Type **Display Label**.

11. Click **OK**.

    Your dialog box should appear similar to the following dialog box.



    When your dialog box appears similar to the sample dialog box, save your edits and exit the designer window.

12. Choose **File → Save**.

13. Choose **File → Exit**.

You can type comments with information for the dialog box, purpose of creation, and so on in the Comments area of the Designer dialog box.

Editing Object Sources

NOTE: You should read the section "Objects in Applix Builder" in Chapter 2 for an understanding of Applix Builder object method usage before completing this section.

After you create the application dialog box, you need to edit the object method source of the toggle button in the First dialog box. Editing object methods and using the Edit Source window are covered in Chapter 3, "Editing Object Methods". A copy of this application is available in the */install_dir*/axdata/eng/Demos/MainDemo/bld_demo/intro directory. Save the intro_first.ab file in your directory to run and edit the example.

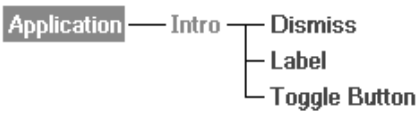
Applix Builder applications are event-driven; clicking on a button or typing in an entry area initiates a course of action. If the event operations are not defined for an object, no object action occurs in response to the event. You will program the changed_event of the toggle button to carry out the application actions.

You want the changed_event of the toggle button to perform the following actions:

- Display or hide the toggle button title, dpending on the state of the toggle button
- Display or hide the label in the dialog box, dpending on the state of the toggle button

To program the changed_event in the First dialog box:

1. Click on the Application object in the Browser window.
2. Choose **View → Expand All**.
3. Click on **Toggle Button** in the Browser window.



4. Click on the Edit Source icon. An Edit Source window appears.

Type the following lines in the Edit Source window after the @@@ OBJECTS ELF directive.

```
var object sibling
set initialize_event
        sibling = this.sibling@("Label")
        sibling.is_hidden@ = FALSE
        this.is_on@ = TRUE
endset

set changed_event(state)
        case of state
        case 0  'Toggle Button off
                sibling.is_hidden@ = TRUE
                this.title@ = "Hide Label"
        case 1  'Toggle Button on
                sibling.is_hidden@ = FALSE
                this.title@ = "Display Label"
        endcase
endset
```

5. Choose **File → Compile & Save**.

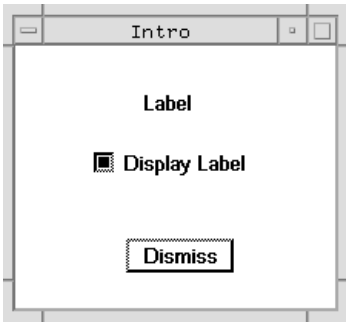
6. Choose **File → Exit**.

After the object method source compiles without errors you are ready to run the application.

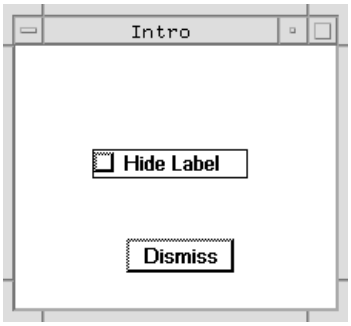
Running an Application

You can run an application from the Builder main menu. You can run an application while you are working on it, allowing you to make changes and fine tune your application. To run an application:

1. Click on the **Run** button on the Builder main menu. The main dialog box appears.
2. Click on the toggle button to turn it on or off.



Toggle Button On



Toggle Button Off

When the toggle button is on, the label is displayed and the toggle button title is Display Label. When the toggle button is off the label is hidden and the toggle button title is Hide Label.

3. Click on **Dismiss** to close the dialog box and exit the application.

Saving and Exiting

Choose File → Save to save the current application. The first time you save an application, the Save As dialog box appears so you can specify a name, location, format, and permissions. To close the application, choose File → Exit. If you made changes since the last save, you get a chance to: save the edits; discard them but still close the application; or cancel the exit and return to the application without making any of the changes.

If you are just trying to open another application, this message appears when the application you are closing has changed since the last save. You must indicate what should be done with those edits (Save, Discard, or Cancel) before the Open option displays.

See "Application File Options" in Chapter 4, "Using the Browser Tool" for information on opening, saving, and exiting an application.

Building an Application with Two Dialog Boxes

This section shows how to build an advanced introductory sample application. The application consists of a two dialog boxes. The first dialog box takes a name from an entry area and displays the name in the second dialog box. This section covers:

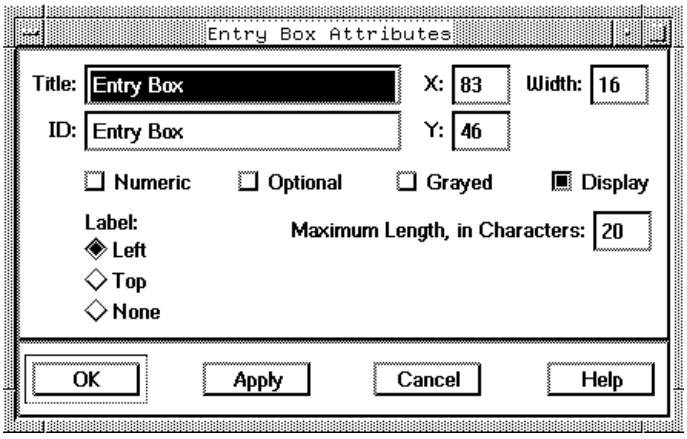
- Creating dialog boxes
- Editing object sources
- Running the application
- Saving and exiting the application

Creating Dialog Boxes

Create a dialog box with the Designer tool. Every application requires at least one dialog box. This sample application requires two dialog boxes. To create the dialog boxes:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type **First** in the Dialog Boxes entry area.
3. Click on **Create**.
4. Choose **Main** in the Type option.
5. Click on **Edit**. The designer window appears.
6. Select the Dismiss button that is automatically inserted in the dialog box.
7. Choose **Edit → Delete Selected**.
8. Choose **Controls → Entry Box** and click in the designer window.

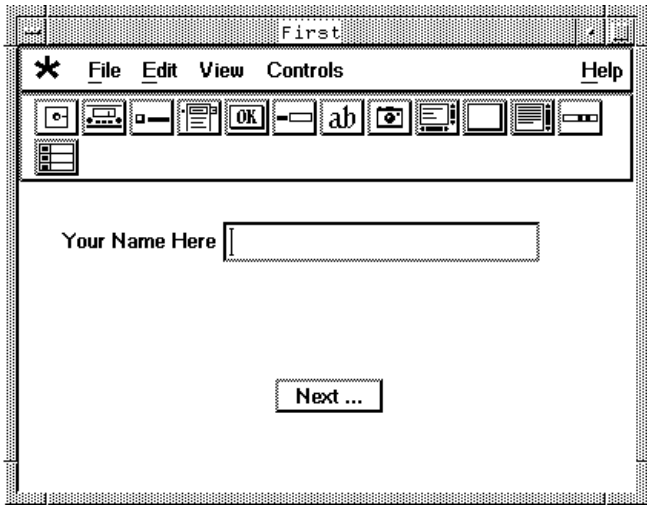
   An entry box appears in the designer window. Click and drag on the handles to resize the entry box in the dialog box.
9. Double-click on the entry box to change the entry box attributes. The Entry Box Attributes dialog box appears.



10. Double-click in the Title entry area.
11. Type **Your Name Here**.
12. Click **OK**.
13. Choose **Controls → Push Button** and click in the designer window.

    A push button appears in the designer window. Click and drag on the handles to resize the push button in the dialog box.
14. Double-click on the push button to change the push buttons attributes. The Button Attributes dialog box appears.
15. Double-click in the Title entry area.
16. Type **Next ...**.
17. Click **OK**.

    Your dialog box should appear similar to the following dialog box.



When your dialog box appears similar to the sample dialog box, save your edits and exit the designer window.

18. Choose **File → Save**.
19. Choose **File → Exit**.

You can type comments with information for the dialog box, purpose of creation, and so on in the Comments area of the Designer dialog box. After completing your comments, you need to create the second dialog box for the application. To create the second dialog box:

1. Double-click in the Dialog Boxes entry area of the Designer dialog box.
2. Type **Second** in the Dialog Boxes entry area.
3. Click on **Create**.
4. Choose **Normal** in the Type option.
5. Click on **Edit**. The designer window appears.

6. Choose **Controls → Label** and click in the designer window.
7. Double-click on the label to change the entry box attributes. The Label Attributes dialog box appears.
8. Double-click in the Title entry area.
9. Type **Hello, my name is:**.
10. Double-click in the ID entry area.
11. Type **Fixed**.
12. Click **OK**.
13. Choose **Controls → Label** and click in the designer window.
14. Drag the label below the first label in the dialog box.
15. Position the Dismiss push button that is automatically inserted in to the dialog box when it is created.
16. Click **OK**.

    Your dialog box should appear similar to the following dialog box.

When your dialog box appears similar to the sample dialog box, save your edits and exit the designer window.

17. Choose **File → Save**.

18. Choose **File → Exit**.

You can type comments with information for the dialog box, purpose of creation, and so on in the Comments area of the Designer dialog box.When you have completed your comments, click Dismiss to dismiss the Designer dialog box.

Editing Object Sources

NOTE: You should read the section "Objects in Applix Builder" in Chapter 2 for an understanding of Applix Builder object method usage before completing this section.

After you create the application dialog boxes, you need to edit the object method source of the buttons in the First dialog box. Editing object methods and using the Edit Source window are covered in Chapter 7, "Editing Object Methods". A copy of this application is available in the /*install_dir*/axdata/eng/Demos/MainDemo/bld_demo/intro directory. Save the intro.ab file in your directory to run and edit the example.

Applix Builder applications are event-driven; clicking on a button or typing in an entry area initiates a course of action. If the event operations are not defined for an object, no object action occurs in response to the event. You will program the clicked_event of the buttons in the sample dialog boxes to carry out the application actions.

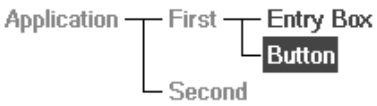
You want the clicked_event of the push button in the First dialog box to perform the following actions:

- Get the name typed into the entry area

- Get a handle to the label in the Second dialog box
- Set the label text to the input name, or, if no name is given, set the label text to "No name given"
- Open the Second dialog box
- Close the First dialog box

To program the clicked_event in the First dialog box:

1. Double-click on **Application** in the Browser window.
2. Double-click on **First** in the Browser window.
3. Click on **Button** in the Browser window.



4. Click on the **Edit Source** icon. An Edit Source window appears.

Type the following lines in the Edit Source window after the @@@ OBJECTS ELF directive.

```
#include "builder_.am"
set clicked_event
        var object sibling, otherdbox, label, app
        var value

        sibling = this.sibling@("Entry Box")
        /* Get value from entry box   */
        value = sibling.value@
        app = this.application@
        otherdbox = app.child@("Second")
        label = otherdbox.child@("Label")
```

```
        /* Set label in Second to value from First entry box */
        IF value = NULL
                label.value@ = "No name given"
        ELSE
                label.value@ = value
        /* Set Second dbox type to Main, then open */
        otherdbox.type@ = DBOX_TYPE#MAIN_
        otherdbox.open@("Second")
        /* Change First dbox type to Normal  */
        this.parent@.type@ = DBOX_TYPE#NORMAL_
        /* Now we can close First (Main) dbox
          without closing application       */
        this.parent@.close@
endset
```

5. Choose **File → Compile & Save**.

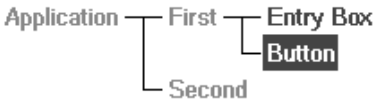
6. Choose **File → Exit**.

The clicked_event gets a handle to the application object because it uses the application object to get a handle to the Second dialog box, open the Second dialog box, and close the First dialog box. The dialog box type of the First dialog box is changed from Main to Normal to allow closing of the dialog box without closing the application. When a Main dialog box is dismissed the application is closed.

You also need to program the terminate_event in the Second dialog box. You need to program the terminate_event to reset the First dialog box as the main dialog box, so that the program will be ready for execution. Code for a clicked_event is automatically inserted into the Dismiss object method source to close the dialog box.

To program the terminate_event in the Second dialog box:

1. If the Application object hierarchy is expanded, click on **Second** in the Browser window, otherwise double-click on **Application**, then click on **Second**.



2. Click on the Edit Source icon. An Edit Source window appears.

Type the following lines in the Edit Source window after the @@@ OBJECTS ELF directive.

```
#include "builder_.am"
set terminate_event
        var object app, otherdbox

        app = this.application@
        otherdbox = app.child@("First")
        otherdbox.type@ = DBOX_TYPE#MAIN_
        /* Change Second dbox type back to Normal  */
        this.type@ = DBOX_TYPE#NORMAL_
endset
```

3. Choose **File → Compile & Save**.
4. Choose **File → Exit**.

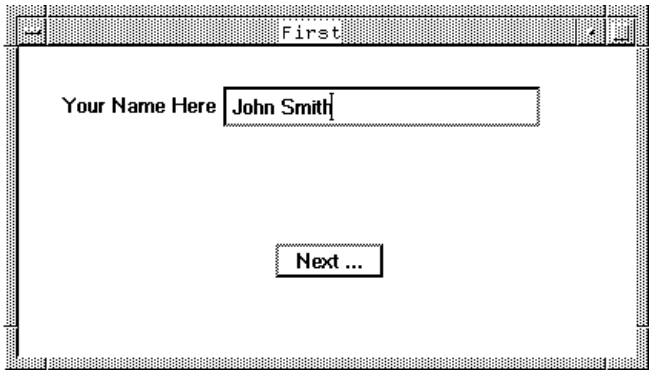
After programming the terminate_event in the Second dialog box you are ready to run the application.

Running an Application

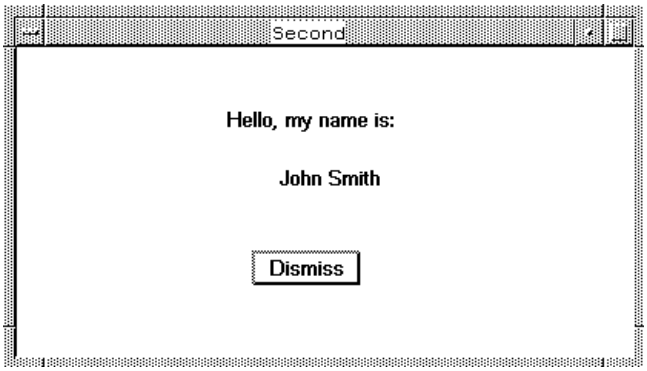
You can run an application from the Builder main menu. You can run an application while you are working on it, allowing you to make changes and fine tune your application.

To run an application:

1. Click on the **Run** button on the Builder main menu. The First dialog box appears.
2. Type a name in the Your Name Here entry area.



3. Click on **Next ....** The Second dialog box appears, and the First dialog box is dismissed.



4. Click on **Dismiss**. The Second dialog box is dismissed and the application is closed.

Saving and Exiting

Choose File → Save to save the current application. The first time you save an application, the Save As dialog box appears so you can specify a name, location, format, and permissions.

To close the application, choose File → Exit.

If you made changes since the last save, you get a chance to: save the edits; discard them but still close the application; or cancel the exit and return to the application without making any of the changes.

If you are just trying to open another application, this message appears when the application you are closing has changed since the last save. You must indicate what should be done with those edits (Save, Discard, or Cancel) before the Open option displays.

See "Application File Options" in Chapter 4 of the *Applix Builder User's Guide*, "Using the Browser Tool" for information on opening, saving, and exiting an application.

5 Using Real Time Methods

A real time data feed provides a continuous stream of changing information, such as financial market information or data recorded by scientific instrumentation. Use a real time data feed to access, analyze, and publish large volumes of continually changing data in your Applix Builder application. This chapter covers the following topics:

- Real Time overview
- Creating Real Time objects
- Programming Real Time objects
- Using HistoricalDataClass methods

Real Time Overview

A real time data feed is composed of records, fields, and a gateway to the data distribution feed. In Applix Builder a real time data feed is composed of:

- A RealtimeGatewayClass object
- One or more RealtimeRecordClass objects
- Optional RealtimeTemplateClass objects

A RealtimeGatewayClass object handles the connection to the data distribution feed. The associated RealtimeGatewayClass methods:

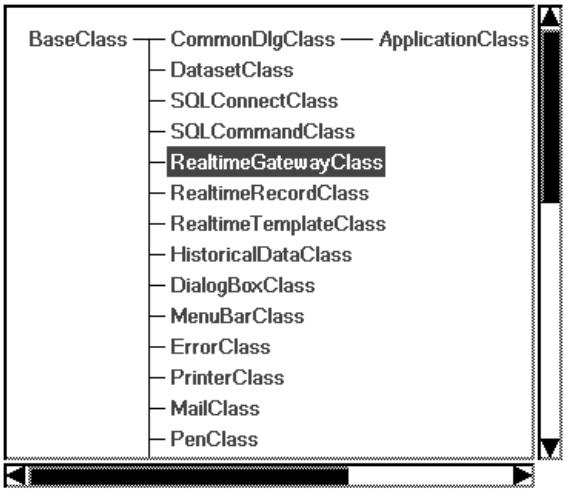
- establish the gateway name
- connect and disconnect the gateway to the data distribution feed
- add, remove, register, and unregister records with the gateway
- add and remove record field templates
- implement an RTSQL query

A RealtimeRecordClass object handles the individual record information. The associated RealtimeRecordClass methods:

- set the service name and record ID for the record
- register and unregister the record with the gateway
- set the fields or field template for the record
- publish record values to the data distribution feed

A RealtimeTemplateClass object provides field information for RealtimeRecordClass objects. Use a template to define a common set of fields you can use across multiple records. A template reduces the duplication of defining fields for individual records. The associated RealtimeTemplateClass methods add and delete fields and converters in the template.

Fields in records or templates can have field converters or field publish converters associated with them. A field converter is a macro or method that converts a value from the data feed before it is displayed in the application. A field publish converter is a macro or method that converts a value received from the application before it publishes the value to the data feed.



Creating Real Time Objects

The RealtimeGatewayClass objects must be added as a child of the top-level ApplicationClass object. RealtimeGatewayClass objects cannot be children of a DialogBoxClass object. RealtimeRecordClass and RealtimeTemplateClass objects must be added as children of a RealtimeGatewayClass object. You can create Real Time objects:

- Using the Source tool
- Adding Real Time objects
- Creating Real Time objects programmatically

See "Creating a Real Time Data Feed" in Chapter 3 of the *Applix Builder User's Guide*, "Using the Source Tool", for information about creating a real time data feed using the Source tool.

Adding Real Time Objects

The RealtimeGatewayClass objects must be added as a child of the top-level ApplicationClass object. To add a RealtimeGatewayClass object to an application:

1. Select the top-level ApplicationClass object in the Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the object name in the Object entry area.
4. Choose **RealtimeGatewayClass** with the Class option.
5. Click on **OK**.

The RealtimeGatewayClass object is added as a child of the ApplicationClass object. After you add a RealtimeGatewayClass object you need to add records and, if desired, templates to the gateway. RealtimeRecordClass and RealtimeTemplateClass objects must be added as children of a RealtimeGatewayClass object. To add a record or a template to the gateway:

1. Select the RealtimeGatewayClass object in the Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the object name in the Object entry area.
4. Choose **RealtimeRecordClass** or **RealtimeTemplateClass** with the Class option.
5. Click on **OK**.

The record or template is added as a child of the gateway. See Chapter 4, "Using the Browser Tool", for more information on adding objects to an application. You need to define the gateway, record, and template information in the respective object method sources to produce a functional real time data feed. See "Programming Real Time Objects" later in this chapter for information on setting the required real time information.

Creating Real Time Objects Programmatically

You can create a real time object programmatically in order to provide a dynamically produced data feed. For example, you may have a gateway object to which you add or remove records as needed. To create a real time object programmatically, you need to:

1. Create the object in an object method source with the object_create@ macro.

2. Assign the object as a child of an object in the application. If you create a RealtimeGatewayClass object, it must be a child of the ApplicationClass object. If you create a RealtimeRecordClass or RealtimeTemplateClass object, it must be a child of a RealtimeGatewayClass object.

You need to ensure that the real time object is removed from the application with the object_destroy@ macro upon exiting the application, even if the application exits because of an error.

See the next section, "Programming Real Time Objects", for an example of creating a RealtimeRecordClass object programmatically.

Programming Real Time Objects

The methods available in the Real Time classes provide great flexibility in programming the data feed actions. The primary areas of interest are creating records, subscribing to the data feed, and publishing information to the data feed.

RealtimeGatewayClass Methods

If you create and define a gateway with the Edit Real Time Gateway dialog box, as described in the "Creating a Real Time Data Feed" section of Chapter 5, the necessary gateway information is set. You can use the RealtimeGateway methods to change an existing gateway, or to set the parameters of a gateway you add as an object or create programmatically.

The set method gateway@ sets the name of the gateway, the Real Time engine, in the RealtimeGateway object. The connect@ and

disconnect@ methods handle gateway connection to the data feed. The add_record@ and add_template@ methods add records and field templates to the gateway, while the delete_record@ and delete_template@ methods delete records and field templates from the gateway. The record registration methods are discussed in the next section, "RealtimeRecordClass Methods".

RealtimeRecordClass Methods

If you create and define a record with the Edit Real Time Gateway and Edit Real Time Record dialog boxes, as described in the "Creating a Real Time Data Feed" section of Chapter 5, the necessary record information is set. If you add a record as an object, or create it programmatically, then you need to define the record ID, the service name of the records, and the fields or template used by the record. You also need to register the record with the gateway. For example, the following initialize_event in a DialogBoxClass object creates a RealtimeRecordClass object programmatically, adds it to an existing RealtimeGatewayClass object, sets the record information, and registers the record with the gateway. The record uses fields defined in a RealtimeTemplateClass object.

```
set initialize_event
        var object rt_data
        var object gate
        /* create record  */
        rt_data = object_create@("RealtimeRecordClass",
                                 "currency")
        /* get handle to gateway, add record as child  */
        gate  = this.application@.child@("RT Gate")
        IF NOT gate.is_connected@
                gate.connect@
        gate.add_child@(rt_data)
```

```
        /* set record info, register record with gateway  */
        rt_data.record_id@ = "DEM="
        rt_data.service_name@ = "IDN_SELECTFEED"
        rt_data.template_name@ = "Currency Template"
        rt_data.register@
endset
```

The next example adds a record using the RealtimeGatewayClass methods. This example and the previous example produce the same result, adding a RealtimeRecordClass object as a child of the gateway object.

```
set initialize_event
        var object gate
        /* get handle to gateway, add record as child  */
        gate  = this.application@.child@("RT Gate")
        IF NOT gate.is_connected@
                gate.connect@
        gate.add_record@("currency","IDN_SELECTFEED",
                        "DEM=", NULL,"Currency Template")
        gate.register_record@("currency")
endset
```

The first example can be used if you need a handle to the record for reference in other methods, or if you change the gateway associated with the record. The second example adds and registers the record with the gateway without external references.

You can register a record from either the gateway or the record. The RealtimeRecordClass methods register@ and unregister@ handle the registration of the record with the gateway object, which should be the record's parent object. The RealtimeGatewayClass methods register_record@ and unregister_record@ handle the record registration on an individual basis, while the register_all@ and

unregister_all@ methods handle registration of all records in the gateway in one step.

The RealtimeRecordClass contains several methods for publishing information to the data feed. Use the methods that begin with the string publish_ to publish individual or groups of records to the data feed.

The field_converter@ method sets a converter macro or method used by a field to convert values from the data feed for display in a dialog box control. The field_publish_converter@ method sets a converter macro or method used by a field to convert values from the dialog box control to the data feed.

Connecting Records to Dialog Box Controls

You can use the Connector to connect RealtimeRecordClass objects to dialog box controls. You can also use the methods defined in the ControlClass to connect a record to a dialog box control. All dialog box control classes descend from the ControlClass, so all the ControlClass methods are available in the inherited control classes.

Connecting with the Connector

The data_source_type@ method sets the type of data source connected to the dialog box control. To define the data source as Real Time, set the value to DATA_SOURCE#REALTIME, as defined in the *install_dir*/axdata/elf/builder_.am file, where *install_dir* is the directory where Applix Builder is installed.

After you define the data source type, use the rt_gateway@, rt_record_name@, and rt_field_name@ methods to define the real time information to associate with the control. A TableClass object can display multiple records and fields, use the rt_record_list@ and

rt_field_list@ methods, instead of the rt_record_name@, and rt_field_name@ methods, to associate multiple records and fields with a table. The following example is an initialize_event for an EntryField-Class object, associating the DSPLY_NAME field of the currency record to the entry field.

```
#include "builder_.am"
set initialize_event
        this.data_source_type@ =
          DATA_SOURCE#REALTIME
        this.rt_gateway@(this.application@.child@("RT Gate"))
        this.rt_record_name@ = "currency"
        this.rt_field_name@ = "DSPLY_NAME"
endset
```

A control's real time data source is the gateway object of the record to which it is connected. There are several advantages to connecting records to dialog box controls with the Connector. If you connect a record to a dialog box control with the Connector, the following update_event code is automatically inserted into the object method source of the control.

```
set update_event
        if not is_null@(this.data_source@)
                this.value@ = this.data_source_value@
endset
```

This update_event is necessary to get the updated values from the real time data feed. Another advantage of using the Connector is automatic registration of the control with the record. When a control is connected to a record with the Connector, the record keeps track of all controls connected to it. When a field value in the record changes, the record automatically calls the update_event of the control connected to the record field.

Connecting with Methods

If you want a dialog box control to continually receive updated real time values, but you do not use the Connector to connect records to dialog box controls, then you must:

- Add the update_event code to the control's object method source
- Add code to the record's data_update_event to call the control's update_event.

The data_update_event receives an array of field indices to the record fields that changed. You can use the indices to call an individual control update_event, instead of broadcasting to all controls. The following are an initialize_event and data_update_event you can use in a RealtimeRecordClass object to update controls with the most recent real time value. The order of the fields in this record are DSPLY_NAME, BID, and ASK.

```
var object kids
set initialize_event
        var object dbox
        dbox = this.application@.child@("Dialog1")
        kids[0] = dbox.child@("Name entry")
        kids[1] = dbox.child@("Bid entry")
        kids[2] = dbox.child@("Ask entry")
endset
set data_update_event(fields)
        var i
        for i = 0 to ARRAY_SIZE@(fields)-1
                kids[field[i]].update_event
        next i
endset
```

The RealtimeRecordClass also contains methods that reference the fields in the record by index. The index methods contain the string by_dex. Use these methods to optimize the efficency of your code. For example, to get the value for one field, you can use the get method display_value_by_dex@ with the field index to efficiently retrieve the value. The get method display_value@, which takes a field name, must internally resolve the field before returning the field value, while the get method display_value_array@ returns all field values in an array, which you must go through to get one value.

Implementing an RTSQL Query

You can use an RTSQL record to retrieve database information in real time to use in your application. The record uses the rtsql gateway to implement the query. The RealtimeRecordClass contains methods, prefaced with rtsql_, that you can use to set and verify the RTSQL query information. See "Entering an RTSQL Query" in Chapter 3, "Using the Source Tool", of the *Applix Builder User's Guide* for information about the elements of an RTSQL query and how to create an RTSQL query with the Builder interface.

Real Time Examples

The livewire application represents a robust application using real time classes. Choose Tools → Sample Applications to open the application in the livewire directory with the Directory Displayer.

Introductory real time examples are available in the rt directory. The examples are based upon the sample rtdemo gateway provided with Applix Builder.

Using HistoricalDataClass Methods

Some data feeds may have historical data that you can retrieve through their service. Use the HistoricalDataClass methods to create a historical data query and retrieve the information to use in your application.

The use the query@ method to initiate a query. The set method will update the controls associated with the query. The get method will update the controls and return the results in a two-dimensional array. The query@ method takes the following arguments:

service	The name of the data feed service for example IDN_SELECTFEED.
record	The name of the record, for example APLX.O.
period	The string for the time period. The valid time period strings are: Daily Weekly Monthly Quarterly Yearly
start_date	The start date for the search, for example 6/1/95.
end_date	The end date for the search. If a date is not supplied and not set with set method end_date@ the current date is used.

tossNullFlag	A Boolean value, set to TRUE to exclude NULL points, FALSE to include NULL points.
date_order	The order in which the historical data is returned. Pass 0 for ascending date order(older to new), or 1 for descending date order (new to older).
showFieldFlag	A Boolean value, set to TRUE to show field names, FALSE to display only data.

You can use other methods in the HistoricalDataClass to set individual arguments in the query. The passed arguments to the query@ method override any arguments set by the previous query or by the individual methods. If any of the arguments are not supplied or are set to NULL the values set by previous queries or by the individual methods are used.

The set method widgets@ associates a control with the historical data query. The method takes a reference to a control as an argument. TableClass and ChartClass controls are the only controls that can receive the results of a historical data query.

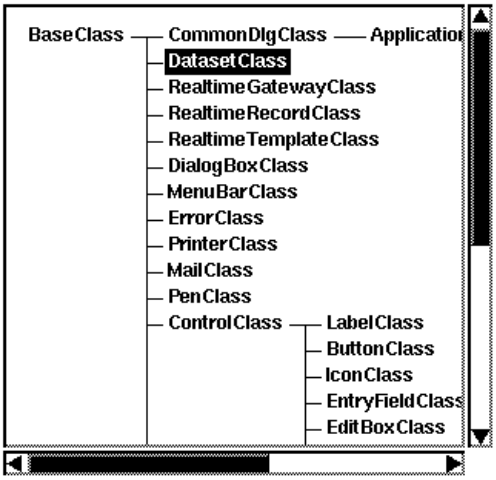
6 Using Database Methods

A data set provides database access for an Applix Builder application. Use the DatasetClass methods to create, retrieve, edit and update database information in your Applix Builder application. You can also use the SQLConnectClass and SQLCommandClass methods to provide simple, direct database editing This chapter covers the following topics:

- DatasetClass overview
- Creating a DatasetClass object
- Querying a data set
- Editing database information
- Using SQLCommandClass and SQLConnectClass methods

DatasetClass Overview

A data set is an SQL query on one or more databases. A data set is represented in Applix Builder as a DatasetClass object. You can create DatasetClass objects as discussed in Chapter 3 of the *Applix Builder User's Guide*, "Using the Source Tool", then use the DatasetClass methods to modify the object. You can also add DatasetClass objects in the Browser window, or create DatasetClass objects programmatically. The next section, "Creating a DatasetClass Object", describes the process of creating a data set.



Creating a DatasetClass Object

The DatasetClass object must be added as a child of the top-level ApplicationClass object. DatasetClass objects cannot be children of a DialogBoxClass object. You can create a DatasetClass object:

- Using the Source tool
- Adding a DatasetClass object
- Creating a DatasetClass object programmatically

See Chapter 3 in the *Applix Builder User's Guide*, "Using the Source Tool", for information on creating a data set using the Source tool.

Adding a DatasetClass Object

To add a DatasetClass object to an application:

1. Select the top-level ApplicationClass object in the Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the object name in the Object entry area.
4. Choose **DatasetClass** with the Class option.
5. Click on **OK**.

The DatasetClass object is added as a child of the ApplicationClass object. See Chapter 2 in the *Applix Builder User's Guide*, "Using the Browser Tool", for more information on adding objects to an application. You need to define the login information and query conditions for the object to produce a useful data set. See "Querying

and Editing with a Data Set" later in this chapter for information on establishing login information and query conditions for the object.

Creating a DatasetClass Object Programmatically

You can create a DatasetClass object programmatically to provide a dynamically produced data set. For example, you may want a data set that only exists for the duration of the application execution, or you may have changing database conditions that require drastically different data sets. To create a DatasetClass object programmatically, you need to:

1. Create a DatasetClass object in an object method source with the object_create@ macro.
2. Assign the object as a child of the ApplicationClass object.

You need to ensure that the DatasetClass object is removed from the application with the object_destroy@ macro upon exiting the application, even if the application exits because of an error.

For example, the following lines of source code can be placed in the object method source of any object in the application to create and destroy a DatasetClass object named parts.

```
set initialize_event
        var object parts_data
        parts_data = object_create@("DatasetClass","parts")
        this.application@.add_child@(parts_data)
endset

set terminate_event
        var object part
        part = this.application@.child@("parts")
        IF object_exists@(part)
```

```
                object_destroy@(part)
endset

get error_event(obj)
        var object part
        part = this.application@.child@("parts")
        IF object_exists@(part)
                object_destroy@(part)
        return(FALSE)
endget
```

You need to define the login information and query conditions for the object to produce a useful data set. See the next section, "Querying a Data Set", for information on establishing login information and query conditions for the object.

The set method bootstrap@ allows you to create either simple or complex data sets programmatically. You can use the method in two different manners.

If you use the method with arguments, the method will put the data set into a functional state, filling in the tables, column list, joins, and other information in a minimal way. The query columns are initialized as all columns in the table. This approach cannot be used with more than one table. If a join is required, you need to use an advanced approach.

If you use the method without arguments, you create a data set where you need to define the data set information, such as tables, joins, and query conditions. This approach is necessary to create a complex data set programmatically.

The reason for this distinction is to provide the means for establishing a simple data set without limiting the functionality available to advanced programmers.

The following is an example of creating a simple data set:

```
set clicked_event
        var tabname
        var object dataset
        /*Create the DatasetClass object  */
        tabname = "axparts"
        dataset =
          object_create@("DatasetClass","table_object")
        /*Create the data set  */
        dataset.bootstrap@("Sybase",
          tabname, "SampleDB", null, "SYBASE")
endset
```

To create a complex data set programmatically you need to define the data set information, such as tables, joins, and query conditions. You need to define the database, establish a connection, then set the data set information. The following is an example of creating a complex data set:

```
#include "dbase_.am"
set changed_event
        var object dataset, table
        var vendor, user, pwd, dbase, machine, server
        var format arrayof sql_table_info@ tables
        /*Get a handle to a TableClass object  */
        table = this.sibling@("Table")
        /*Create the DatasetClass object  */
        dataset =
          object_create@("DatasetClass","axparts_object")
        /* initialize data set  */
        dataset.bootstrap@
        /* Set data set info, establish the connection */
```

```
        vendor = "Sybase"
        user = "applix"
        pwd = "helpme"
        dbase = "SampleDB"
        machine = null
        server = "SYBASE"
        dataset.define_database@(vendor,
          user,pwd,dbase,machine, server)
        dataset.connect@
        /* data set tables */
        tables[0].name = "applix.axparts"
        tables[0].uid = "applix.axparts"
        tables[0].colnames =
                { "part_num", "part_name", "part_descr" }
        dataset.tables@ = tables
        /* data set columns */
        dataset.displayed_columns@ = tables[0].colnames
        /* Query database, put values in a table*/
        dataset.requery@
        table = dataset.all_values@
endset
```

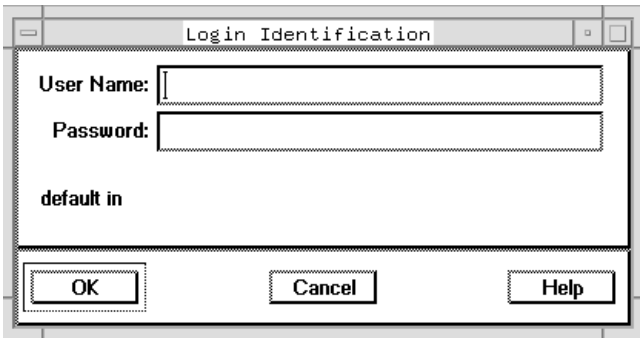
The get method all_values@ returns the values retrieved by the data set query as a two-dimensional array. The returned information is prepared to be placed directly in a TableClass object. The method takes an array of column names as an optional argument. If the column names are passed as an argument, the information in the columns is returned for each row. If no argument is passed to the method, all the information in the displayed columns for the data set is returned.

Querying a Data Set

This section describes how to use the DatasetClass methods to query a database. Querying a database involves setting the login information for the database, choosing the tables to query, and establishing query conditions. Chapter 3 in the *Applix Builder User's Guide*, "Using the Source Tool", describes the steps to establish a database query with the Applix Builder Source tool.

Connecting

To establish a connection to the database you need to choose the database and, if required, set the login name and password. The define_database@ method sets the database vendor, database, and host machine information. You can also set the login name and password with this method, or use the login@ method to set the information separately. The login_global@ method sets the default user name and password for all database login. Use the login_dlg@ method to set login information with the Login Identification dialog box.



After you define the databases and set the login information, use the connect@ method to establish a connection to the database. After you establish a connection you need to establish the data set tables. Use the tables@ method to set the tables. The tables@ method takes as arguments an array of table information, an array of join information, and a Boolean value that determines if the data set information is reset. The tables@ method by default resets all the data set information, except for join, conditions, and having information, unless the reset flag is set to FALSE. All selected columns are removed, except for columns that are expressions or column names that match column names in the selected tables.

You can use the join@ method to create a table join in the data set. If you define a table join you do not need to set the tables@ method.

Setting Conditions and Querying

After you set the table or join information in the table you need to set the query conditions. Use the conditions@ method to establish a set of one or more query conditions. If you have an SQL statement that you want as the query source, use the source@ method and pass the SQL statement, in an array of strings, as the argument to the method.

If you need to execute an SQL statement to alter database information or return data use the exec_direct@ methods. The set version of the method only executes an SQL statement, while the get version executes an SQL statement and returns the results.

The data set automatically requeries on any change in the query conditions. Set the auto_query@ method to FALSE to allow changes to query conditions without performing a query. Use the requery@ method to execute a query when auto_query@ is disabled.

Use the position@, decrement_position@, and increment_position@ methods to move through the data set rows and set the current position in the data set.

Use the disconnect@ method to disconnect the data set from the database when you are finished querying and editing with a data set.

Editing Database Information

Editing the database information includes inserting, deleting, and updating database records. The model for editing is that all inserts, updates, and deletes of rows in the current data set are held locally until the edits are committed to the database. The two-step process of applying edits, then committing edits to close a transaction, is applicable for data set connections with all databases that have transactions.

Set the is_editing_enabled@ method to TRUE to enable editing of the rows in the data set. Each row retrieved by the data set must be unique for trouble-free editing. If there are not enough columns in the query to uniquely identify each row, then errors may occur when edits are applied to the database.

As you edit, a record of your changes is kept internally in the data set, indicating inserted and deleted rows, and changed columns in updated rows. When you change a value in an existing row, the row becomes locked in the database. All edits are kept internally until you apply edits.

Use the current_record_value@ method to set a value in a field of the current row. The method takes the column name and the new value as arguments. To set a field value in other rows, use the value@ method. The method takes the column name, record number, and new value as arguments, so you can set information in rows other than the current row.

Use the edit_delete@ and edit_insert@ methods to, respectively, delete and insert rows in the database.

Use the edit_apply_only@ method to apply the edits in the data set to the database. All the changes made to the current data set are applied to the database in this operation. If an error occurs while the edits are applied, an error is thrown and the row where the error occurred becomes the current row. If the row was deleted it becomes undeleted. Remaining edits for rows after the row containing the error remain pending. At this point one of the following steps must be taken to resolve the error:

- rollback all edits with edit_rollback@
- commit the applied edits with edit_commit_only@
- unedit the error row with edit_undo@
- apply edits (to try again)
- fix the error row and retry the apply
- make other edits

The edit_commit_only@ method completes the transaction for the DBMS by commiting the edits. The transaction is started when one of the following occurs:

- a row is locked
- the first edit_apply_only@ or edit_commit@ method is called.

Using SQLConnectClass and SQLCommandClass Methods

The SQLConnectClass and SQLCommandClass methods allow you to directly edit a database programmatically. Use the classes together to provide complete database editing functionality in your application. This section cover the following topics:

- Establishing a connection
- Editing the database

Establishing a Connection

Use the SQLConnectClass methods to establish the connection to the database. After you create an SQLConnectClass object, you can use the set method open_gateway@ to open the gateway to the database. After you open the gateway, use the set method connect@ to connect to the database.

After you connect to the database, you can use the SQLCommandClass methods to retrieve and edit database information. You can also use the SQLConnectClass exec_direct@ methods to execute an SQL statment directly. You can use the set method to add or remove tables, or add or delete rows to tables. You can use the get method if the SQL statement returns information.

When you are finished using the database methods, you can disconnect from the database and close the gateway connection. Use the set methods disconnect@ and close_gateway, to perform the respective actions.

Editing the Database

Use the SQLCommandClass methods for advanced database manipulation after you establish the connection to the database with the SQLConnectClass methods. Use the SQLConnectClass set method command@ to create an SQLCommandClass object to use with the SQLConnectClass object in your application.

Use the set method prepare@ to preapre an SQL statement for a database query. After you prepare a statement, use the set method open_cursor@ to open a cursor to the database table. The cursor can be either an editing cursor or a query cursor. Use the pos@ and next_pos@ methods to set or get the current or next cursor position in the rows meeting the SQL statement criteria.

To edit information, use the row editing functions. The set method insert_row@ inserts a row before the current row, while the set method delete_row@ deletes the current row. The set method update_column@ updates a column in the current row with the passed information, while the set method unedit_row@ cancels all edits in the current row.

When you are finished editing the database, use the set method apply_edits@ to apply the edits to the database. After you apply the edits, use the SQLConnectClass methods commit@ or rollback@ to either commit the edits or cancel the edits, then close the transaction. After commiting or cancelling the edits, use the set methods close_cursor@ and unprepare@ to close the cursor to the database and unprepare the database.

The following code example is used to query the axparts table and put the information in a table control.

```
objvar object supplier
objvar vals
objvar object stmt
set initialize_event
```

```
        var query, head, val2
 /* Create SQLConnectClass object   */
        supplier =
object_create@("SQLConnectClass","parts")
/* add as child of ApplicationClass object   */
        this.application@.add_child@(supplier)
        supplier.open_gateway@("star","Sybase")
        supplier.connect@("SERV1","sampledb")
        stmt = supplier.command@()
        vals = NULL
        stmt.prepare@("select supplier_num, company
from.axsupplier", FALSE)
        stmt.open_cursor@(NULL,FALSE)
        WHILE NOT is_null@(stmt.next_pos@)
                vals[array_size@(vals)] = stmt.row@
        wend
endset

set update_event
        this.value@ = vals
endset

set terminate_event
        stmt.close_cursor@
        stmt.unprepare@
        IF object_exists@(stmt)
                object_destroy@(stmt)
        supplier.disconnect@
        supplier.close_gateway@
endset
```

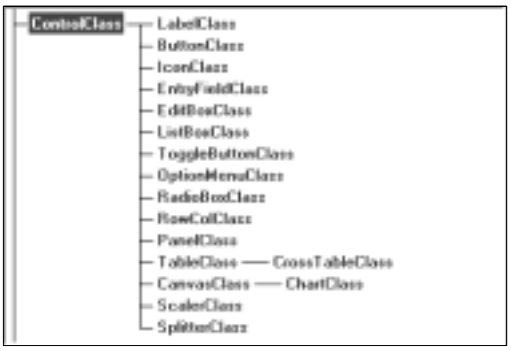
7 Using Dialog Box Controls

This chapter describes the common methods for dialog box controls. Most of these methods are defined in the ControlClass. This chapter covers the following topics:

- Setting data source attributes
- Setting display position and appearance
- Setting colors and fonts
- Using control events

Dialog Box Control Overview

All dialog box controls inherit from the ControlClass. The Control-Class is an abstract class, you cannot instantiate an object based directly on the ControlClass. The ControlClass contains the methods common to all controls.



This chapter will also cover methods that are not defined in the ControlClass, but behave similarly in most of the dialog box controls. For information about a specific control refer to the following chapters in this manual.

Help for ControlClass methods and events is available through the Class Browser dialog box. Choose View → Classes to use the Class Browser dialog box.

Setting Data Source Attributes

You can associate various types of data sources with a dialog box control, to display database, Real Time, or other types of information. The Connector provides an interface for connecting dialog box controls. The Connector, however, is not always the best means for associating a control with a data source.

If your application needs to dynamically change the association between a data source and a control during application execution, then you need to use the ControlClass methods to define the changes.

This section covers the following ControlClass methods:

data_set@
data_set_field_name@
data_set_field_name_list@
data_set_field_value@
data_source@
data_source_is_connected@
data_source_macro@
data_source_type@
data_source_value@
display_map_macro@
rt_field_name@
rt_field_value@
rt_gateway@
rt_record_name@
validator_macro@

Data Source Methods

Use the data_source_* methods to set and determine the data source associated with the control. The get method data_source_is_connected@ returns a Boolean value indicating if a data source is connected to a control. The method returns TRUE if a

data source is connected to the control, otherwise the method returns FALSE.

After you determine if a data source is connected to a control, you can get a reference to the data source with the get method data_source@. The method returns the data source object associated with the control. You can use a reference to the data source to update or change the data source, such as requerying a database or unregistering a Real Time record.

Use the set method data_source_type@ to define the data source type associated with a control. The valid source types, as defined in the header file builder_.am, are:

- DATA_SOURCE#NONE None
- DATA_SOURCE#MACRO Macro
- DATA_SOURCE#DATA_SET Data set
- DATA_SOURCE#REALTIME Real Time

A data source macro is a macro that you define that is capable of setting or returning information. If you define the data source of a control as a macro, use the set method data_source_macro@ to set the macro called as the data source.

Data Set Methods

Use the data_set* methods to assign data sets to controls. Data sets are usually created with the Source tool.

Use the set method data_set@ to set the data set used by the control. The method takes a data set object as an argument.

Use the set method data_set_field_name@ to assign a data set field to a control. The method takes a field name as an argument. Use this

method with entry boxes, labels, and other such controls that only display a single piece of information.

Use the set method data_set_field_names@ to assign muliple data set fields to a control. The method takes an array of field names as an argument. Use this method with tables, edit boxes, and other such controls that can display multiple pieces of information.

Use the set method data_set_field_value@ to set the field of the data set to the passed value. Use the get method data_set_field_value@ to get the current value of the filed in the data set. You can use these methods cooperatively to verify and set values in your application.

Real Time Methods

Use the rt_* methods to associate a Real Time data source with a control. To associate a Real Time data source with a control, you need to define:

- the gateway for the Real Time data feed
- the record in the Real Time gateway
- the field in the Real Time record

Use the set method rt_gateway@ to assign the gateway to the control. The method takes a RealtimeGatewayClass object as an argument.

After you assign the gateway, you can assign the record to the control with the set method rt_record_name@. The method takes the name of the record as an argument. The name of the record must be a valid record registered with the gateway.

After you assign the record, you can assign a field of the record to the control with the set method rt_field_name@. The method takes the

name of the field as an argument. Valid field names are determined by the data feed vendor.

After a Real Time data source is fully assigned to a control, you can use the set and get methods rt_field_value@ to set and retrieve the Real Time value of the field in the record. Use of the set method may be limited by the restrictions of data publishing to the data feed at your site.

Display Maps and Validators

Display maps and validators are macros or methods that you use to manipulate data source information. You need to define in the body of the macro or method exactly how the information is to be manipulated. Use the set method display_map_macro@ to set the display map for a control. The display map manipulates the information from the data source before it is displayed in the control.

Use the set method validator_macro@ to set the validator for a control. The validator manipulates the information from the control before sending it to the data source.

These set methods take the name of a macro or method as an argument. If you use a method as a display map or validator, the method must be a get method, a method that returns information. If you use a mcro as a display map or validator, the macro name must be preceded by the @ character, otherwise the display map or validator is considered a get method.

See Chapter 5 in the *Applix Builder User's Guide*, "Using the Connector Tool", for more information about using display maps and validators.

Setting Display Position and Appearance

All dialog box controls have position attributes. The ControlClass contains methods to set control position, as well as set the display characteristics of the control.

This section covers the following ControlClass methods:

display@	title@
display_suspend@	x_pos@
is_grayed@	y_pos@
is_hidden@	

Setting Position

When you use the Dialog Box editor window to design your dialog box, you can adjust the position of a control by dragging it with the mouse or by setting the control attributes. You can also use the set methods x_pos@ and y_pos@ to set the x- and y-position of the control. The position of a control in a dialog box is measured, in pixels, from the upper-left corner of the dialog box.

Setting Title

Use the set method title@ to set the title of a control. Controls such as labels, buttons, radio buttons, toggle buttons, and option buttons have titles that are visible in the dialog box. Other controls, such as panels and list boxes, do not have visible titles; these titles are used only for reference.

Graying

Use the set method is_grayed@ to gray a control. Set the method to TRUE to make a control gray. Set the method to FALSE to make a control appear in its normal state. A grayed control is unavailable in the dialog box, you cannot directly choose or otherwise change the state of a grayed control.

Normal Grayed

Hiding and Displaying

Use the set method is_hidden@ to hide a control in the dialog box. For example, instead of graying a feature that doesn't apply in a certain context, you can hide a control to avoid user confusion. Set the method to TRUE to hide the control. Set the method to FALSE to display the control in the dialog box.

Use the set methods display@ and display_suspend@ together to control the display of controls in a dialog box. Set display_suspend@ to TRUE to suspend updates to the control's display. For example, if you suspend the display of a label, the label title, font, font size, and so on will not change, even if it is set explicitly in another control's method. Set the method to FALSE to make the control update with any change.

Use the display@ method to update controls with suspended displays. You can broadcast the display@ to update all controls at one time in the dialog box, instead of updating each control individually for each separate change. For example, use the following lines in a DialogBox-Class object to broadcast a display@ to all controls in the dialog box.

```
var object kids
kids = this.all_children@
kids.*display@
```

Setting Colors and Fonts

Most controls in a dialog box have color or font attributes that you can set. Because all controls do not have the same set of attributes, the color and font methods are not defined in the ControlClass, but locally in each control class. This section will cover the color and font methods in general, use the Class Browser dialog box to determine the specific color and font methods available in a control class.

This section covers the following color and font methods:

*_color@	*_font_attrs@
*_color_cmyk@	*_font_bold@
*_color_is_workspace@	*_font_italic@
*_color_name@	*_font_name@
*_color_rgb@	*_font_size

The font methods are prefaced with either text or title. The color methods are prefaced with either control, text, or title.

The preface control refers to the work area of the control. Use the control_* methods to set the colors of the control work area.

The preface text refers to the text of the choices in a radio button group or option menu, or the text that appears in a list box, edit box, or entry

box. The text settings apply to all text that appears in the control, you cannot apply different text settings in a single control.

The preface title refers to the text that is the title of the control, such as the title of an option button group, the text in a push button, or the text of a label.

Color Methods

You can use the various color methods to set the text, title, or control colors. The colors available in an application are dependent upon the display monitor, the color map, and your preferences settings.

Use the set method *_color@ to set a color depending on the color type. You can use the method to set the color with:

- RGB values
- HSB values
- CMYK values
- a color name
- the default widget color
- the work area color

All of the other color methods are specific instances of the *_color@ method. For example, the lines

```
this.control_color@(6, 0, 221,221,0)
```

and

```
this.control_color_cmyk@(0,221,221,0)
```

both set the control color to red using the CMYK values for red.

Font Methods

You can use the various font methods to set the text or title font attributes. The font attributes available for an application are dependent upon the display monitor and defined font mapping in the operating environment.

Use the set method *_font_attrs@ to set multiple font attributes for the text or title. The method takes an argument in font_attrs_info@ format. The format is defined in the builder_.am header file as follows.

font_name	The font name.
font_size	The font point size.
bold	The font bold state as a Boolean value. TRUE sets the font to bold.
italic	The font italic state as a Boolean value. TRUE sets the font to italic.

The other font methods set the individual font attributes. Use the individual attribute methods if you are only changing one attribute, otherwise it is more efficient to use the *_font_attrs@ method.

Using Control Events

All objects in an Applix Builder application have, at a minimum, the following events that you can program:

- error_event
- initialize_event
- terminate_event
- time_out_event

In addition to these events, controls have, at a minimum, two events that you can program:

- resize_event
- update_event

The resize_event is called when the dialog box containing the control is resized. The event is passed four parameters:

- The new dialog box
- the new dialog box height
- the original dialog box width
- the original dialog box height

You can program a resize_event to re-position the control in the dialog box. For example, if want a control to always be positioned 20 pixels from the bottom and 30 pixels from the right edge of a dialog box, you would program a resize_event as follows:

```
resize_event(new_width, new_height, old_width, old_height)
        this.x_pos@ = new_width - 30
        this.y_pos@ = new_height -20
endset
```

The update_event is called when a dialog box update@ or update_children@ method is called. The event is first called after the dialog box is displayed. Program the update_event to set the control properties that you want updated, or to verify that the control properties are within your defined parameters.

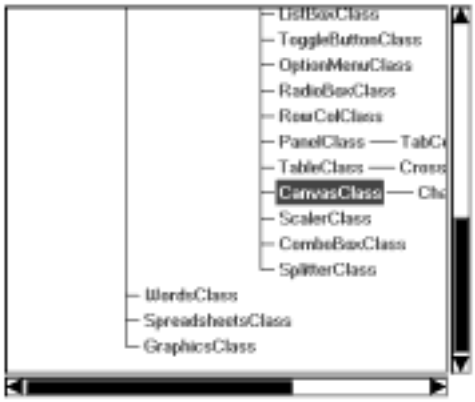
8 Using CanvasClass and PenClass methods

A canvas provides a drawable area in the dialog box. Use a PenClass object and the CanvasClass methods to draw text and graphical objects upon the canvas. Vertical and horizontal scroll bars allow you to move to different areas of the canvas. Refer to XWindows information on graphics context (gc) for an understanding of how to use a PenClass object. This section covers the following topics:

- CanvasClass and PenClass overviews
- Creating CanvasClass object
- Drawing on a canvas
- Using insets

CanvasClass and PenClass Overviews

Use CanvasClass methods to define the drawable area in a dialog box for text and images. The CanvasClass methods define the type of image to draw, the area of the canvas to display, and the color of the canvas. The PenClass methods define the text, line, color, and fill attributes of text and images drawn in the canvas.



Methods and Events

The following CanvasClass and PenClass methods and events are discussed in this chapter:

background_color_cmyk@	font_size@
button_press_event	foreground_color_name@
clear_area@	initialize_event
create_inset@	inset_file@
dashes@	inset_type@
draw_inset@	keyboard_event
draw_line@	line_style@
draw_text@	measure_inset_object@
expose_event	motion_event
font_name@	scroll_event
font_name@	text_width@

Creating a CanvasClass Object

A CanvasClass object exists in a dialog box. The PenClass object should be a child of the CanvasClass object so that it is initialized every time the CanvasClass object is initialized. A PenClass object is

functional if it is added as a direct child of the application, but its initialize_event is only called once, when the application is started. A PenClass object cannot be a direct child of a DialogBoxClass object.


Designer icon

To create a canvas in a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Canvas** in the Dialog Box Editor window.
7. Click in the dialog box to place the canvas.
8. Resize the canvas and set the canvas attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a canvas. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information on creating and editing dialog boxes.

To add a PenClass object as a child of the CanvasClass object:

1. Select the CanvasClass object in the Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the object name in the Object entry area.
4. Choose **PenClass** with the Class option.
5. Click on **OK**.

The PenClass object is added as a child of the CanvasClass object. See Chapter 2 in the *Applix Builder User's Guide*, "Using the Browser Tool", for more information on adding objects to an application.

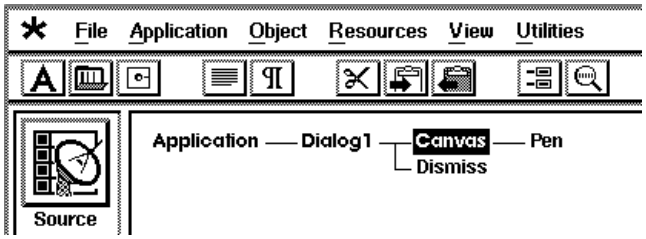
Drawing on a Canvas

After you establish CanvasClass and PenClass objects you need to determine the type of text or images you want to display in the canvas. CanvasClass set methods have no effect until the canvas is displayed. Call set methods in the various CanvasClass events after the canvas is displayed.

The CanvasClass has many events that respond to mouse, keyboard, and scrolling actions. The canvas allows you to create an interactive application with few components.

Canvas Example

The following is an example of a canvas where you can draw a free flowing line. The example contains a CanvasClass object named Canvas, a PenClass object named Pen, and a ButtonClass object named Dismiss.



The source code for the PenClass object Pen contains the following lines in the initialize_event to set the foreground color, background color, line style to dashes, and a space of 20 pixels between dashes.

```
@@@ OBJECTS
#include "elfpen_.am"
set initialize_event
        this.foreground_color_name@ = "Blue"
        /* set background color to White  */
        this.background_color_cmyk@(0,0,0,0)
        this.line_style@ = XVAL_LineOnOffDash
        this.dashes@ = 20
endset
```

The source code for the ButtonClass object Dismiss contains the following lines in the clicked_event to close the application.

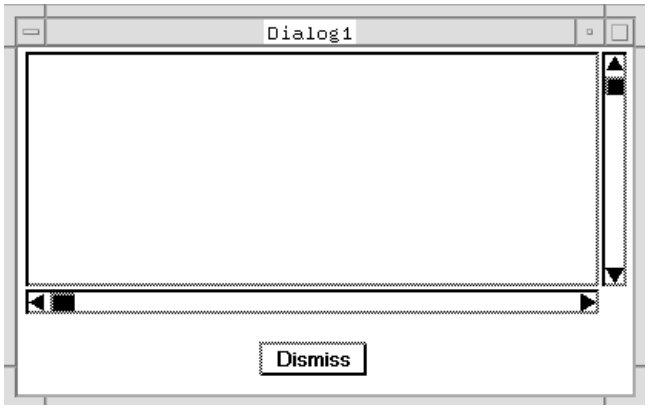
```
@@@ OBJECTS
set clicked_event
        this.application@.quit@
endset
```

The source code for the CanvasClass object Canvas contains the following lines for an initialize_event, button_press_event, motion_event, and a user-defined method named draw_line. The initialize_event gets a handle to the PenClass object. The button_press_event sets the starting point in the canvas. The

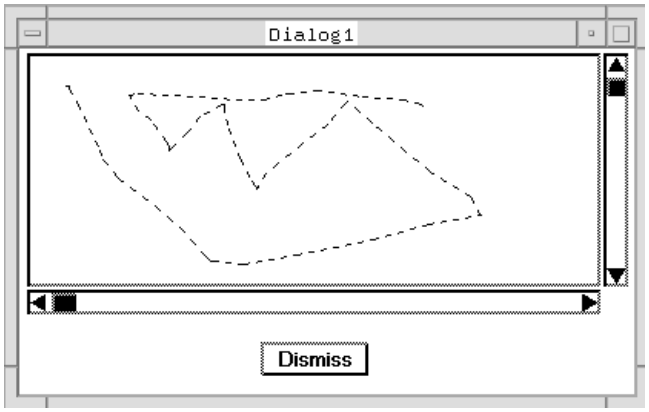
motion_event makes a call to the user-defined method draw_line, then sets the starting point of the next line segment. The user-defined method draw_line calls the CanvasClass method draw_line@ with the PenClass object Pen, the start position defined by the objvar start, and the end position passed from the motion_event.

```
@@@ OBJECTS
#include "elfpen_.am"
objvar pen
objvar start

set initialize_event
        pen = this.child@("Pen")
endset

set button_press_event(position)
        start = position
endset

set motion_event(position)
        this.draw_line(position)
        start = position
endset

set draw_line(pos)
        this.draw_line@(pen,start,pos)
endset
```

When you run the application, the dialog box is displayed with an empty canvas.



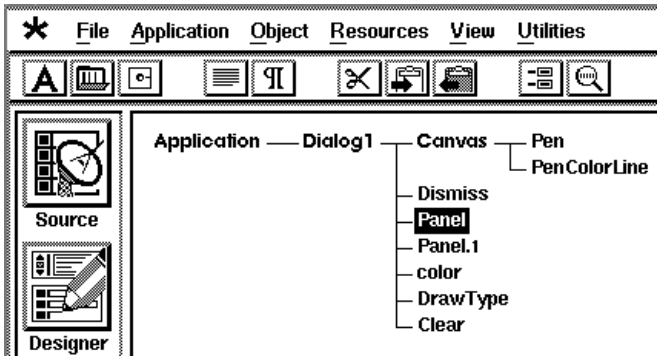
When you press down the left mouse button and drag in the canvas a dashed line is drawn on the canvas. When you release the mouse button the line drawing stops.



If you use the scroll bars to move to a different area of the canvas the material on the canvas is cleared when it moves out of the display area. You need to program the expose_event and scroll_event to maintain the existing information, or to display new information.

Advanced Canvas Example

The following is an advanced example that builds upon the first example. The application allows you to draw a line or text in the canvas. The application also has a pop up menu that is accessed with a right mouse button press. The application contains a CanvasClass object named Canvas, two PenClass objects, two ButtonClass objects, two RadioBoxClass objects, and two PanelClass objects.



The source code for the PenClass object Pen contains the following lines in the initialize_event to set the font to Palatino, the font size to 14 point, and background color to white.

```
@@@ OBJECTS
#include "elfpen_.am"
set initialize_event
        this.font_name@(XVAL_font_palatino)
        this.font_size@(14)
        this.background_color_cmyk@(0,0,0,0)
endset
```

The source code for the PenClass object PenColorLine contains the following lines in the initialize_event to set the foreground color,

background color, line style to dashes, and a space of 20 pixels between dashes.

```
@@@ OBJECTS
#include "elfpen_.am"
set initialize_event
        this.foreground_color_name@ = "Blue"
        /* set background color to White  */
        this.background_color_cmyk@(0,0,0,0)
        this.line_style@ = XVAL_LineOnOffDash
        this.dashes@ = 20
endset
```

The source code for the ButtonClass object Dismiss contains the following lines in the clicked_event to close the application.

```
@@@ OBJECTS
set clicked_event
        this.application@.quit@
endset
```

The source code for the ButtonClass object Clear contains the following lines in the clicked_event to clear the canvas display area and set the text buffer to NULL.

```
@@@ OBJECTS
set clicked_event
        var object canvas, pen
        sysvar buffer
        canvas = this.sibling@("Canvas")
        pen = canvas.child@("Pen")
        canvas.clear_area@(pen,0,0,canvas.width@,
canvas.height@)
        buffer = NULL
```

```
endset
```

The source code for the CanvasClass object Canvas contains the following lines for an initialize_event, button_press_event, motion_event, keyboard_event, and user-defined methods named draw_line, move_text, set_line, set_text, quit, and clear.

The initialize_event sets the pop up menu information and gets handles to the RadioBoxClass object DrawType and the PenClass objects.

The button_press_event sets the starting point in the canvas and initializes the text buffer variable flag to TRUE.

The motion_event makes a call to a user-defined method, depending on the drawing type selected. If the drawing type is Line the user-defined method draw_line is called, then the starting point of the next line segment is set. If the drawing type is text the user-defined method move_text is called.

The keyboard_event draws text on the canvas as it is typed in, character by character. Each character is placed in a buffer, so that it can be redrawn if the text is moved with the user-defined method move_text. The keyboard_event has checks to start a new line of text if it is near the right edge of the canvas, and to display an info message when the text reaches the lower right corner of the canvas display area.

The user-defined method draw_line calls the CanvasClass method draw_line@ with the PenClass object PenColorLine, the start position defined by the objvar start, and the end position passed from the motion_event.

The user-defined method move_text calls the CanvasClass method draw_text@ with the PenClass object Pen, the position passed from the motion_event, and the characters stored in the variable buffer. The method gives the appearance of dragging text in the canvas.

The user-defined methods set_line, set_text, quit, and clear are methods called by pop up menu choices.

```
@@@ OBJECTS
#include "elfpen_.am"
#include "dialog_.am"

objvar object type
objvar start
objvar penLine
objvar pen
objvar flag
sysvar buffer

set initialize_event
        this.button3_menu_info@ = {
                {"Draw Line", "set_line", "", TRUE},
                {"Draw Text", "set_text", "", TRUE},
                {"Clear Canvas", "clear", "", TRUE},
                {"Exit Application","quit","",TRUE}
        }
        type = this.sibling@("DrawType")
        penLine = this.child@("PenColorLine")
        pen = this.child@("Pen")
        buffer = NULL
endset

set motion_event(position)
        var valType
        valType = type.value@
        case of valType
```

```
            case "Line"
                  this.draw_line(position, penLine)
                  start = position
            case "Text"
                  this.move_text(position, pen)
            endcase
endset


set button_press_event(position)
            flag = TRUE
            start = position
endset


set keyboard_event(key,shifted, controlled)
            var cr, h, b
            var object color, newColor, Text, pen
            /* initialize buffer */
            IF flag
                  buffer = null
            flag = FALSE
/* get handle to pen, set color  */
            pen = this.child@("Pen")
            newColor = this.sibling@("color")
            color[0] = 5
            color[1] = newColor.value@
            pen.foreground_color@(color)
/* if text, convert ascii value to char, add to text buffer, draw in
canvas */
            if type.value@ = "Text"
```

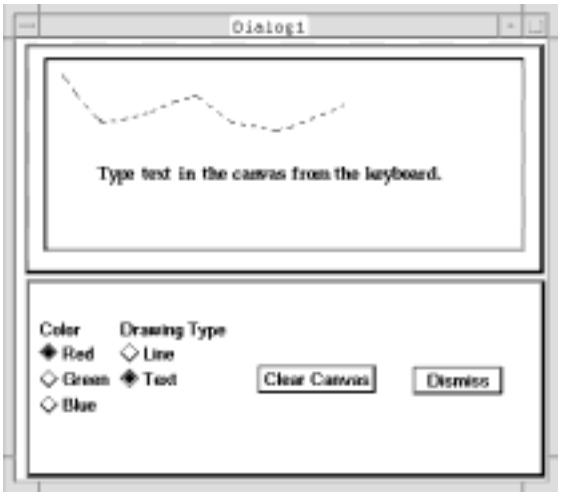
```
		{
		cr = NUM_TO_STRING@(key)
		buffer = buffer ++ cr
		this.draw_text@(pen, start,NULL,cr)
		start[0] = start[0] +3+ (this.text_width@(pen,cr))/2
		/* if near edge of canvas wrap to next line  */
		IF start[0] > (this.width@-10)
		 {
			start[0] = 1
			start[1] = start[1]+ 5 +
(this.text_font_baseline@(pen))
		 }
		IF start[1] > (this.height@-10)
		 {	BEEP@()
			INFO_MESSAGE@("Limit of canvas drawing
area reached")
		 }
		}
endset

set move_text(position, object pen)
		var object color, newColor, Text
/* get color, set by name   */
		newColor = this.sibling@("color")
		color[0] = 5
		color[1] = newColor.value@
		pen.foreground_color@(color)
		this.clear_area@(pen,0,0,this.width@,this.height@)
		start = position

		this.draw_text@(pen,position,NULL,buffer)
```

```
endset


set draw_line(pos, object penLine)
        var object color, newColor, Text
        newColor = this.sibling@("color")
        color[0] = 5
        color[1] = newColor.value@
        penLine.foreground_color@(color)
        this.draw_line@(penLine,start,pos)
endset

/* Methods for right mouse button choices  */
set set_line
        type.value@ = "Line"
endset

set set_text
        type.value@ = "Text"
endset

set quit
        this.application@.quit@
endset

set clear
        this.clear_area@(pen, 0, 0, this.width@,
this.height@)
endset
```

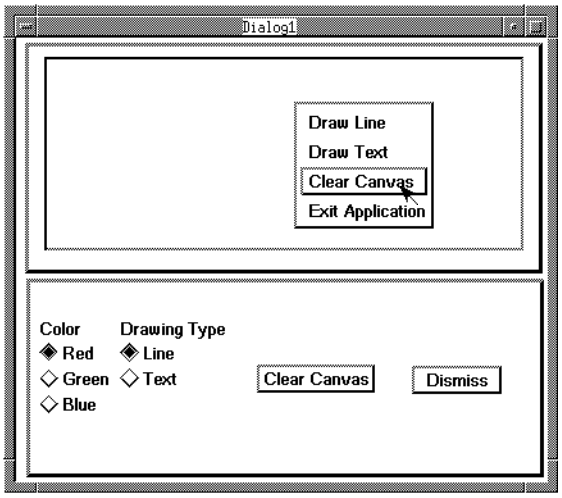
When you run the application, the dialog box is displayed with an empty canvas.

When you select the Line drawing type, press down the left mouse button, and drag in the canvas, a dashed line is drawn on the canvas. When you release the mouse button the line drawing stops.

When you select the Text drawing type, click in the canvas with the left mouse button, and begin typing on the keyboard, text is drawn on the canvas.



When the mouse pointer is in the canvas display area and you press the right mouse button a pop up menu is displayed. When you select a menu item and release the right mouse button the associated menu bar function is called and the pop up menu is disappears.



Using Insets

In addition to text images, you may also need to present complex information in your application on a canvas. You can inset Applix Words, Spreadsheets, or Graphics files in a canvas. An inset can be a native Applixware document, or an imported document type supported by the Words, Graphics, or Spreadsheets filters.

Use the following CanvasClass methods to handle insets on a canvas:

- create_inset@
- draw_inset@
- inset_file@
- inset_type@
- measure_inset_object@

Creating an Inset

Use the create_inset@ or draw_inset@ methods to create the inset in the canvas. Both methods take the same arguments; create_inset@ will create the inset, while draw_inset@ will create the inset and render it on the canvas.

The first argument for either method is the inset information, in ax_inset_object@ format. The format is defined in the *install_dir*/axdata/elf/insets_.am header file. Use the ax_inset_object@ format to define how the inset is rendered on the canvas. For example, you can set the scaling and canvas display characteristics in an initialize_event before drawing the inset on the canvas.

The second argument for create_inset@ and draw_inset@ methods is the specific inset information for the source file. The inset information format depends upon the inset type. The following table defines the inset type, the format, and the header file containing the format. All header files are located in the *install_dir*/axdata/elf directory.

Table 8-1 Inset Source Formats

Application	Format	File
Applix Graphics	gr_inset_source@	graphic_.am
Applix Spreadsheets	ss_inset_source@	spsheet_.am
Applix Words	wp_inset_source@	wp_.am

Defining an Inset

After you create an inset, use the set method inset_file@ to define the file for the inset. Use the corresponding get method to retrieve the file name of the inset. You can use this information, for example, when you want to make the inset editable. The following double_click_event uses the OPEN_DOC@ macro to open the inset in its appropriate Applixware document

```
set double_click_event
        open_doc@(this.inset_file@)
endset
```

Getting Inset Information

Use the get methods inset_type@ and measure_inset_object@ to get detailed information about the inset. The inset_type@ method returns the inset type of the current inset. The valid inset types are:

- 1 Applix Words
- 2 Applix Graphics
- 3 Applix Spreadsheets

You can use this method to confirm loading the proper file type in the inset. In situations where your application is loading non-Applixware files in the inset, you can check the inset type to ensure compatibility with the imported file.

Use the measure_inset_object@ method to get the size information for the inset. The inset returns the size information as a 2-element array. The first element is the measurement units of the inset, in ax_units@ format. The second element is the inset area information, in ax_area@ format. You can use the size information to position the inset on the canvas, or to change the canvas dimensions to accomodate the inset.

Canvas Inset Example

The following example draws an Applix Words inset on the canvas. The initialize_event sets the inset information. The update_event draws the inset on the canvas. The double_click_event opens the inset in the appropriate Applixware document, in this case Applix Words. The expose_event redraws the inset on the canvas, given the new scrolling positions.

```
#include "insets_.am"
#include "wp_.am"
objvar filename
objvar format ax_inset_info@ info
objvar format wp_inset_source@ wp_source
objvar prev_pt
```

```
set initialize_event
        var size
        filename = "/user/applix/sample.aw"
        prev_pt = {0,0}
/* set general inset info  */
        info.clear = true
        info.widget_colors = true
        info.scale_mode = "clip to fit"
  /* set zoom of inset to 45%  of normal size  */
        info.x_zoom = 45
        info.y_zoom = 45
/*  set application-specific inset info  */
        wp_source.area.x = 0
        wp_source.area.y = 0
        wp_source.area.width = this.width@
        wp_source.area.height = this.height@
        wp_source.units.dpi = 75
        wp_source.units.dpu = 1000/75
        wp_source.units.precision = 0
        wp_source.units.name = "pixels"
        wp_source.preserve_whitespace = TRUE
        this.inset_file@(filename)
endset

set update_event
        var size
        var format ax_units@ units
        var format ax_area@ area
        this.draw_inset@(info, wp_source)
        size = this.measure_inset_object@()
        units = size[0]
```

```
            area = size[1]
            if units.name = "mils" {
                        area[0] = area[0]*75/1000
                        area[1] = area[1]*75/1000
            }
 /* set paint dimensions of canvas to inset dimensions  */
            this.paint_width@ = area[0]
            this.paint_height@ = area[1]
endset

set double_click_event
 /* Open source doc in appropriate Applixware application  */
            open_doc@(this.inset_file@)
endset

set expose_event
/* redraw inset on an expose event  */
            var pos
            pos = this.scroll_pos@
            wp_source.area.x = pos[0]
            wp_source.area.y = pos[1]
            wp_source.area.width = prev_pt[0]--
pos[0]+this.width@
            wp_source.area.height = prev_pt[1]--
pos[1]+this.height@
            prev_pt = pos
            this.draw_inset@(info, wp_source)
endset
```

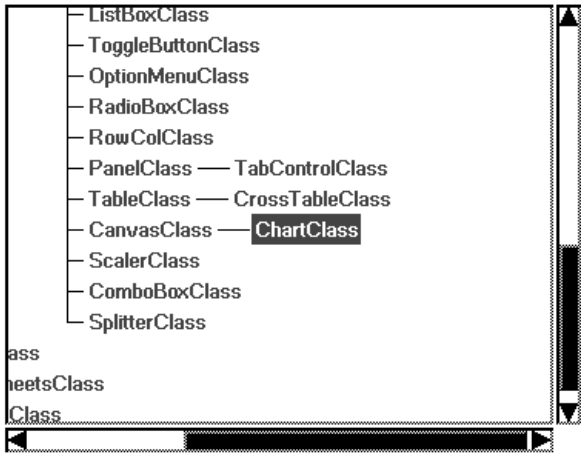
9 Using ChartClass Methods

A chart provides a visual, concrete representation of information. Use ChartClass methods to create and edit charts in your Applix Builder application. This chapter covers the following topics:

- ChartClass overview
- Creating a ChartClass object
- Implementing a chart
- Printing a chart

ChartClass Overview

Use the ChartClass methods to set the data and appearance of a chart in a dialog box. You can also use the ChartClass get methods to retrieve the chart attributes, then use the attributes to display a chart in a Spreadsheets or Graphics document. The ChartClass is based upon the CanvasClass. It also inherits all the methods used to render objects on a canvas.



Some of the ChartClass methods use formats defined in the chart_.am header file, located in the /*install_dir*/axdata/elf directory. Use the methods that take formats as arguments to set many chart attributes at once. Use methods that do not use formats as arguments to set smaller or individual chart attributes.

Creating a ChartClass Object

Creating a ChartClass object is a two-stage process. To create a ChartClass object, you need to:

- Create a canvas in a dialog box using the Designer.
- Change the CanvasClass object to a ChartClass object using the Properties dialog box.

To create a canvas in a dialog box:


Designer icon

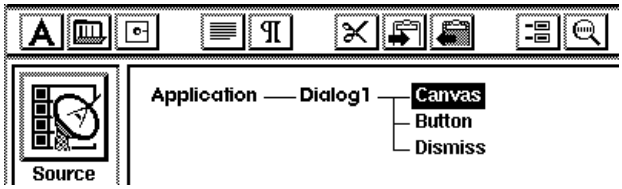
1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Canvas** in the Dialog Box Editor window.
7. Click in the dialog box to place the canvas.
8. Resize the canvas and set the canvas attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.

11. Click on **Dismiss** in the Designer dialog box.

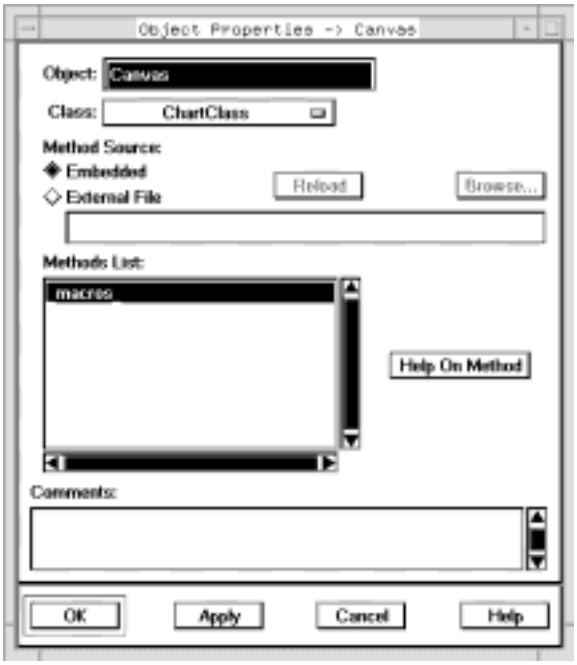
The dialog box is created with a canvas. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information on creating and editing dialog boxes.

After you create the canvas in the dialog box, change the class of the canvas to ChartClass. To change the class of the canvas:

1. Expand the hierarchy in the Browser window to display all children of the dialog box.
2. Select the canvas object in the Browser window.



3. Choose **Object → Properties**. The Object Properties dialog box is displayed.



4. Choose **ChartClass** with the Class option to change the inherited class from CanvasClass to ChartClass.

5. Click on **OK**.

The canvas in the dialog box becomes a ChartClass object. It retains the same appearance in the dialog box. See Chapter 2 in the *Applix Builder User's Guide*, "Using the Browser Tool", for more information on changing object properties.

Implementing a Chart

After you establish a ChartClass object, you need to set the chart information to render a chart in the dialog box. You need to determine how you want to create the chart and set the chart attributes. You can create a chart in response to a ButtonClass clicked_event, in a RealtimeRecordClass data_update_event, or in any other object event. A chart can be as simple or as complex as the information you pass to it and the attributes you set.

The ChartClass set methods have no effect until you create a chart. The minimum information needed to create a chart is the chart type and chart data. The amount of data required for the chart depends on the chart type. The chart types are defined in the ch_type_.am file located in the *install_dir*/axdata/elf directory. Include this file in any object source where you are using defined chart types with ChartClass methods.

Creating a Chart

Use the set method create@ to create a chart in the ChartClass object. After you create a chart, use the ChartClass methods to set additional chart attributes. After all chart attributes are set, use the set method display_chart@ to display the chart in the ChartClass object.

NOTE: You cannot use the display_chart@ method in a ChartClass object's initialize_event. You can use the create@ method and all other methods that set the chart attributes in a ChartClass object's initialize_event.

Setting Chart Attributes

The is_display_on_update@ method sets the update attributes of the chart. If you pass a TRUE argument to the method, the chart is redisplayed whenever the chart data or attributes are changed. When you pass a FALSE argument to the method, changes to the chart are only displayed when you call the display_chart@ method. The default setting is FALSE.

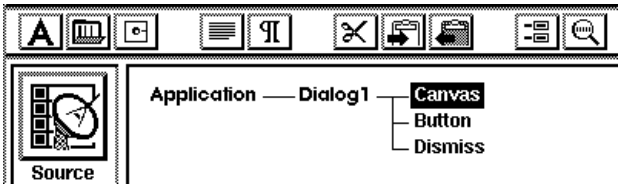
The ChartClass contains the following edit_*_dlg@ methods that display dialog boxes you can use to set chart attributes. Use the dialog boxes to present a consistent graphical interface. You can use the dialog boxes to set multiple attributes at once, instead of setting the attributes individually with multiple methods.

edit_3D_dlg@	Sets 3D chart attributes
edit_axes_dlg@	Sets axes attributes
edit_frame_and_grid_dlg@	Sets frame and grid attributes
edit_layout_dlg@	Sets margins and chart orientation
edit_legend_dlg@	Sets legend attributes
edit_titles_dlg@	Sets title attributes

If you want to change the chart display size in the chart area, use the set method drawing_area@ to set the chart position and dimensions in the drawable chart area.

Chart Example

The following is an example of a column chart with two data groups. The dialog box contains a ChartClass object named Canvas and two push buttons named Button and Dismiss.



The chart's source code contains the following lines in the initialize_event to create a chart.

```
@@@ OBJECTS
#include "ch_type_.am"

set initialize_event
        var data
        data[0,0] = 1
        data[0,1] = 3
        data[1,0] = 2
        data[1,1] = 4
        this.create@(COLUMN,data)
endset
```

The Button push button's source code contains the following lines in the clicked_event to display a chart.

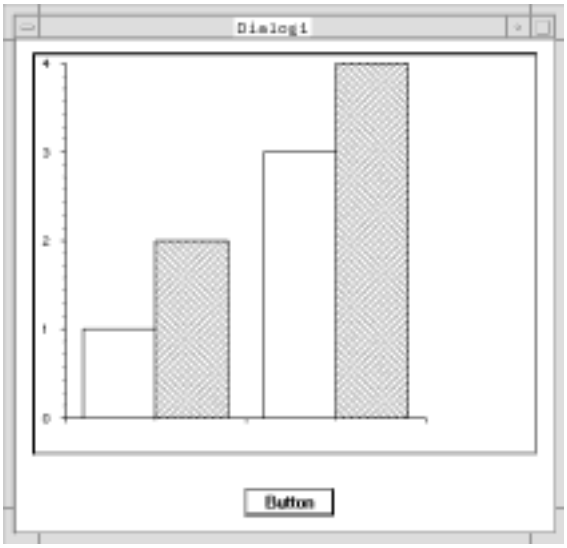
```
@@@ OBJECTS
set clicked_event
        var object chart
        chart = this.sibling@("Canvas")
        chart.display_chart@
endset
```

The Dismiss push button's source code contains the following lines in the clicked_event to close the dialog box.

```
@@@ OBJECTS
set clicked_event
        this.parent@.close@
endset
```

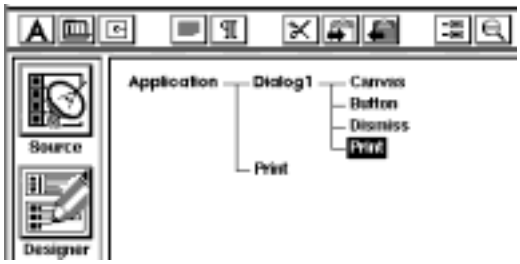
When you run the application, the dialog box is displayed with a blank chart area.

When you click on Button the chart appears in the chart area.



Printing a Chart

After you draw a chart on a canvas you can print the chart using the PrinterClass method draw_chart@. The method takes a handle to the ChartClass object, width, and height of the chart on the printed page as arguments. For example, you can add a Print push button and a PrinterClass object named Printer to the previous example for printing.



The Print push button's source code contains the following lines in the clicked_event to print a chart.

```
@@@ OBJECTS
set clicked_event
        var object chart, p
        /* get handles to chart and printer  */
        chart = this.sibling@("Canvas")
        p = this.application@.child@("Print")
        /* set printer info with setup dialog box  */
        p.print_setup_dlg@
        p.start_job@
        /* draw chart in printable area  */
        p.draw_chart@(chart,5000, 5000)
        p.print@
endset
```

When the Print push button is clicked after drawing the chart, the Print dialog box is displayed with the print options. After the print options are set and the dialog box is dismissed the chart is drawn in the printable area and sent to the printer.

See Chapter 10, "Using PrinterClass Methods", for information about printing application information with PrinterClass methods.

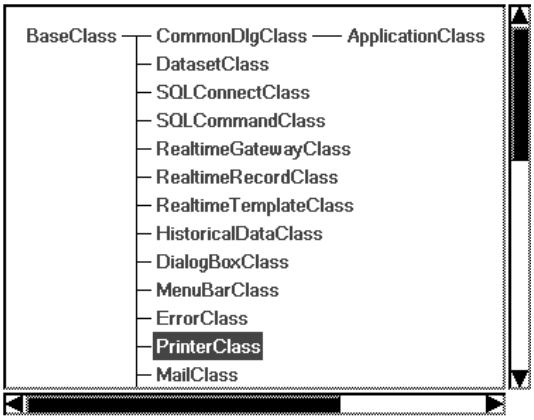
10 Using PrinterClass Methods

Use PrinterClass methods to print the information contained in an application to a PostScript® or PCL5 printer. This chapter covers the following topics:

- PrinterClass overview
- Creating a PrinterClass object
- Printing from an application

PrinterClass Overview

Use the PrinterClass methods to print text and images contained in your application. Use the methods to print directly to a PostScript or PCL5 printer, instead of passing information to a WordsClass, GraphicsClass, or SpreadsheetsClass document, then printing the document.



Help for PrinterClass methods is available through the Class Browser dialog box. Choose View → Classes to use the Class Browser dialog box.

PrinterClass methods that draw text or images in the print space use a PenClass object. The PenClass object font attributes must be set to properly draw in the print area. Setting the PenClass object attributes is covered in the section "Printing From an Application" later in this chapter.

Creating a PrinterClass Object

The PrinterClass object must be added as a child of the top-level ApplicationClass object. The PenClass object should be a child of the PrinterClass object so that it is initialized every time the PrinterClass object is initialized. PrinterClass and PenClass objects cannot be children of a DialogBoxClass object. To add a PrinterClass object to an application:

1. Select the top-level ApplicationClass object in the Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the object name in the Object entry area.
4. Choose **PrinterClass** with the Class option.
5. Click on **OK**.

The PrinterClass object is added as a child of the ApplicationClass object. To add a PenClass object as a child of the PrinterClass object:

1. Select the PrinterClass object in the Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the object name in the Object entry area.
4. Choose **PenClass** with the Class option.
5. Click on **OK**.

See Chapter 2 in the *Applix Builder User's Guide*, "Using the Browser Tool", for more information on adding objects to an application.

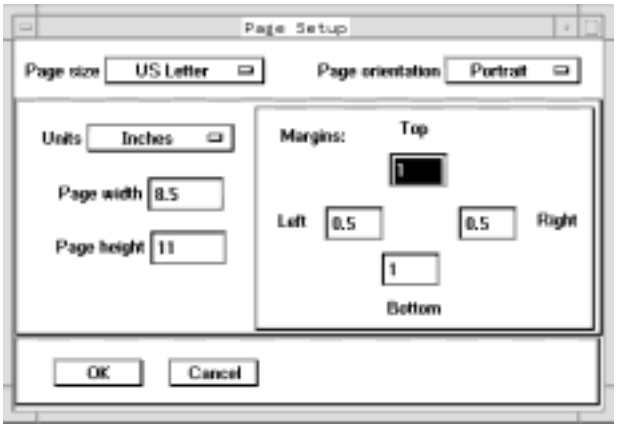
Printing From an Application

After you establish PrinterClass and PenClass objects you can begin printing from your application. All coordinates on the print area are in mils (thousandth of an inch). Points are a two element array of x and y coordinates. The origin for all coordinates, except headers and footers, is relative to the intersection of the top and left margins, which is point (0,0). Start all print jobs with the start_job@ method. The method initializes the printer for a new file and creates the first page of the file. The page dimensions, header text, and footer text must be set before calling the start_job@ method.

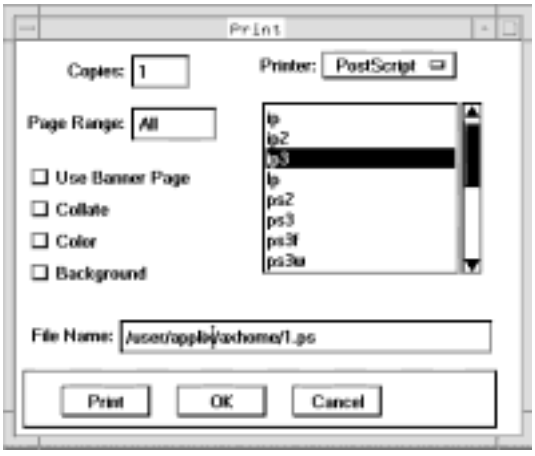
Use headers and footers to place page numbering and other information on the printed page. The header@ and footer@ methods take pen and text arguments for text in the left , center, and right of the header and footer.

Text in headers and footers has special formatting sequences. The sequence #p places the current page number in the header or footer text. The sequence #P places the last page number in the header or footer text. The sequence #d*nnn* places the date and time in the header or footer text. The date and time used is the time at which the start_job@ method is invoked. The string *nnn* is the format for the date, as defined in the datetim_.am file, located in the /*install_dir*/axdata/elf directory. You do not need to include the datetim_.am file in your object method source to place a date and time in the header or footer text.

The page_setup_dlg@ method displays a dialog box you can use to verify and set the page dimensions and margins.



The print_setup_dlg@ method displays a dialog box you can use to verify and set the print file and printer attributes.

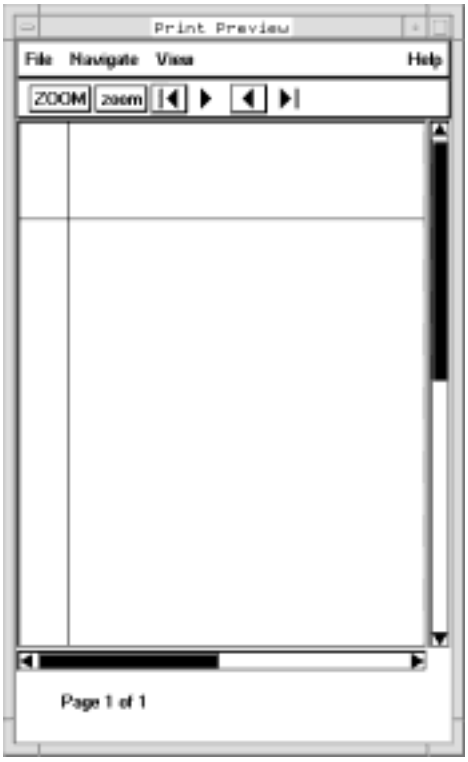


You can also use other methods in the PrinterClass, such as page_metrics@, to set page and printer attributes on an individual basis. After you set the page attributes, invoke the start_job@ method to start the print job.

Starting a Print Job

The start_job@ method takes five optional arguments. Set the first argument to TRUE to display the Print Preview dialog box. You can use the Print Preview dialog box to see the print job layout on the page before you send it to the printer. The additional arguments are only in effect when you display the Print Preview dialog box. The default value for the first argument is FALSE.

The second argument is the title, the passed string becomes the title of the Print Preview dialog box. The third and fourth arguments are, respectively, the x and y coordinates where the dialog box is displayed on screen, relative to the mouse pointer. The fifth argument is the size of the dialog box on screen. The integer passed as an argument is the percentage of the print page displayed. Valid values are from 25% to 100%. The default value is 50%. Margin lines are displayed in the Print Preview dialog box, but the margin lines do not appear in the printed file.



After you set the page attributes and invoke start_job@ you can begin adding material to the print job.

Drawing in the Print Area

Use the draw_ methods to draw in the printable area. You need to set the attributes of the PenClass object before you use it to draw in the printable area. For example, to draw text in the printable area with the draw_text@ method you need to set the font and font size of the PenClass object.

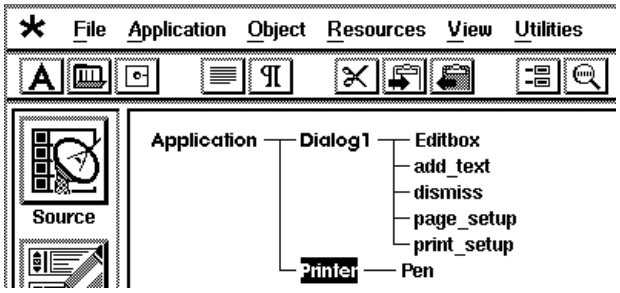
When drawing in the printable area you also need to set the cursor position on the page. The cursor is at position (0,0) for each new page. The cursor position is relative to the printable area, the area within the page margins. Position (0,0) is the intersection of the top and left margins. Cursor position is measured in mils. Use the set method cursor@ to set the cursor at a given point, and use the get method cursor@ to get the current cursor position. The draw_text@ method has an argument to set the auto-advance characteristics of the cursor after you invoke the method.

A print job can consist of multiple pages. Use the new_page@ method to start a new page in the print job. You can set the page dimensions, headers, and footers for each page before you invoke the new_page@ method. A new page is forced if the cursor advances into the bottom margin area as a result of a draw_text@ call.

A new_page_event is called when a new page is started. You can program a new_page_event to set different headers, footers, and page attributes for different pages. The new_page_event is given the page number as an argument. See the sample application later in this section for an example of a new_page_event.

PrinterClass Example

The following is an example of a simple application that prints text added to an edit box. The application contains a PrinterClass object named Printer, a PenClass object named Pen, and a dialog box containing an edit box and four push buttons.



The printer's source code contains the following lines in the new_page_event to place even page numbers in the left corner of the footer, while placing odd page numbers in the right corner of the footer.

```
@@@ OBJECTS
set new_page_event(page_number)
        var object pen
        pen = this.child@("Pen")
        IF (page_number MOD 2 ) = 0
                this.footer@(pen,"#p",NULL,NULL,pen," ")
        ELSE
                this.footer@(pen," ",NULL,NULL,pen,"#p")
endset
```

The pen's source code contains the following lines in the initialize_event to set the text font and size.

```
@@@ OBJECTS
set initialize_event
        this.font_name@ = "Palatino"
        this.font_size@ = 14
endset
```

The push button add_text source code contains the following lines in the clicked_event to draw the text from the edit box in the printable

area, then clears the text from the edit box. If the print job is not started, the start_job@ method is invoked and the Print Preview dialog box is displayed with the title Preview Title Text.

```
@@@ OBJECTS
set clicked_event
        var object edit, print, pen
        edit  = this.sibling@("Editbox")
        print = this.parent@.sibling@("Printer")
        pen   = print.child@("Pen")
        IF NOT print.is_started@
                print.start_job@(TRUE,"Preview Title Text")
        print.draw_text@(pen, edit.value@, 2)
        edit.value@ = NULL
endset
```

The push button page_setup source code contains the following lines in the clicked_event to display the Page Setup dialog box.

```
@@@ OBJECTS
set clicked_event
        var object print
        print = this.parent@.sibling@("Printer")
        print.page_setup_dlg@
endset
```

The push button print_setup source code contains the following lines in the clicked_event to display the Print dialog box.

```
@@@ OBJECTS
set clicked_event
        var object print
        print = this.parent@.sibling@("Printer")
        print.print_setup_dlg@
```

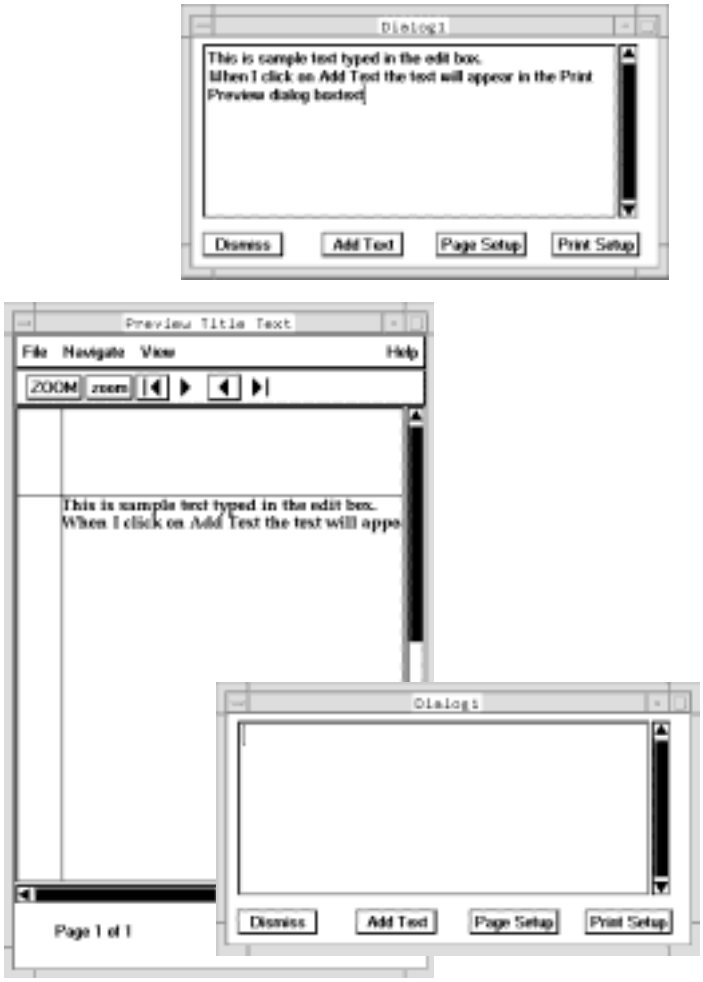
```
endset
```

The push button dismiss source code contains the following lines in the clicked_event to send the print job to the printer and close the application.

```
@@@ OBJECTS
set clicked_event
        var object print
        print = this.parent@.sibling@("Printer")
        print.print@
        this.application@.quit@
endset
```

When you run the application, the dialog box is displayed with an empty edit box.

When you type text in the edit box, then click on Add Text, the Print Preview dialog box is displayed with the text in the printable area.



When you click on Dismiss the print job is sent the printer and the application is closed.

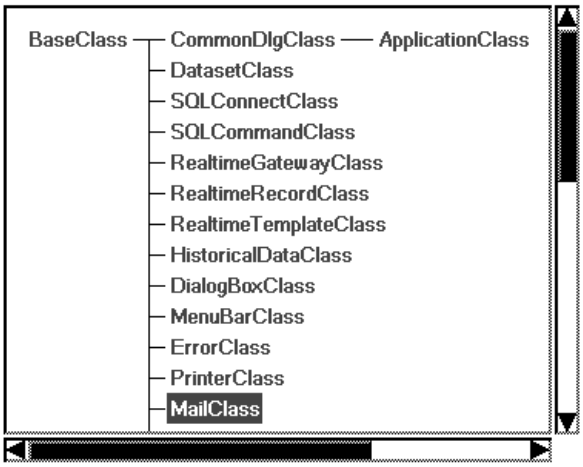
11 Using MailClass Methods

Use MailClass methods to create and send e-mail from an application. This chapter covers the following topics:

- MailClass overview
- Creating a MailClass object
- Mailing from an application

MailClass Overview

Use the MailClass methods to create and send e-mail from your Applix Builder application. You can set the message attributes and send attached files with methods.



Help for MailClass methods is available through the Class Browser dialog box. Choose View → Classes to use the Class Browser dialog box.

Creating a MailClass Object

The MailClass object must be added as a child of the top-level ApplicationClass object. MailClass objects cannot be children of a DialogBoxClass object. To add a MailClass object to an application:

1. Select the top-level ApplicationClass object in the Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the object name in the Object entry area.
4. Choose **MailClass** with the Class option.
5. Click on **OK**.

The MailClass object is added as a child of the ApplicationClass object. See Chapter 2 in the *Applix Builder User's Guide*, "Using the Browser Tool", for more information about adding objects to an application.

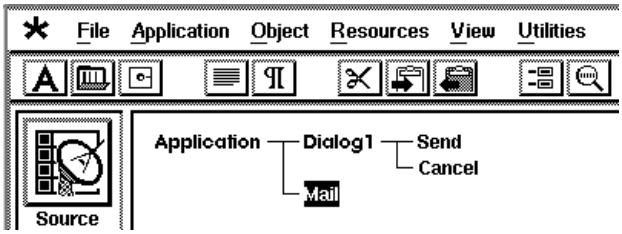
Mailing From an Application

After you establish a MailClass object you can begin mailing messages from your application. Use the MailClass set methods to set the various message attributes, such as recipient, subject, attachments, and message body. The send_mail@ method sends the mail message you create with the other set methods.

The set method is_interactive@ sets the interactive mode for mail messages. If you set is_interactive@ to TRUE a Send Mail dialog box appears before you send the message, loaded with the current message attributes. Use the Send Mail dialog box as a consistent user interface for mail messages in your Applix Builder applications. If is_interactive@ is set to FALSE the message is sent without displaying the Send Mail dialog box. A message must contain at least one recipient and a subject before you can send it. The Send Mail dialog box appears if the message recipients or subject are not set, even if is_interactive@ is set to FALSE.

MailClass Example

The following is an example of a mail application that displays the Send Mail dialog box to send a message. The application contains a MailClass object named Mail, and a dialog box with two push buttons, Send and Cancel.



Mail's source code contains the following lines in the initialize_event to set the interactive mode to TRUE, so that the Send Mail dialog box is displayed when the send_mail@ method is invoked.

```
@@@ OBJECTS
set initialize_event
        this.is_interactive@ = TRUE
endset
```

Send's source code contains the following lines in the clicked_event to send the mail message, which displays the Send Mail dialog box.

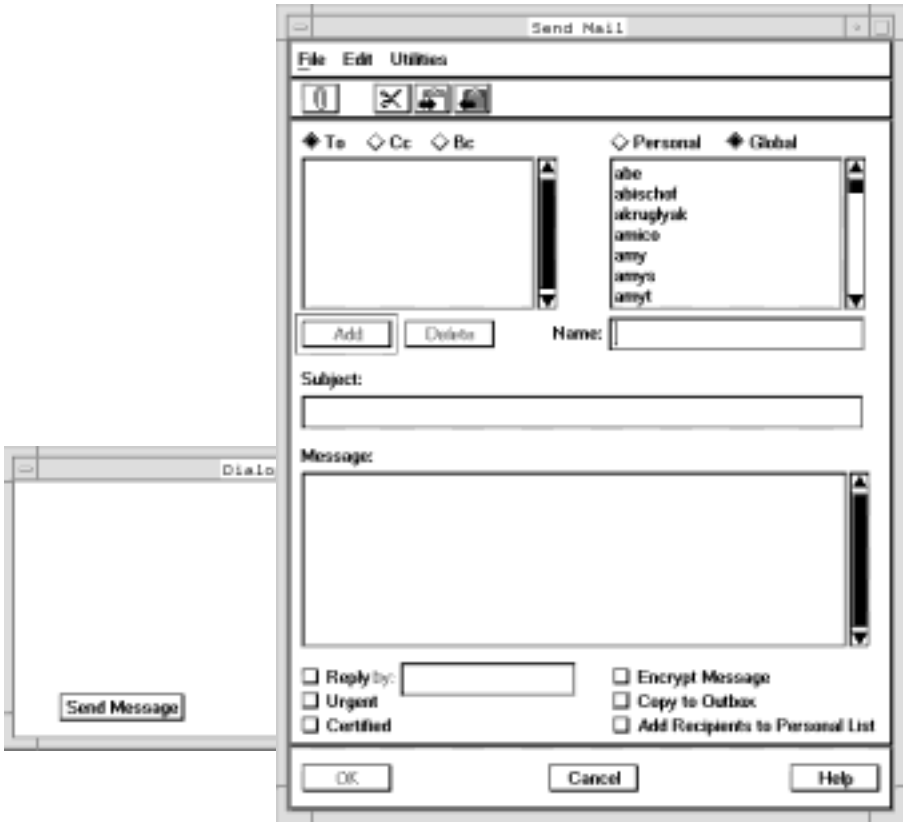
```
@@@ OBJECTS
set clicked_event
        var object msg
        msg = this.application@.child@("Mail")
        msg.send_mail@
endset
```

Cancel's source code contains the following lines in the clicked_event to exit the application.

```
@@@ OBJECTS
set clicked_event
        this.application@.quit@
endset
```

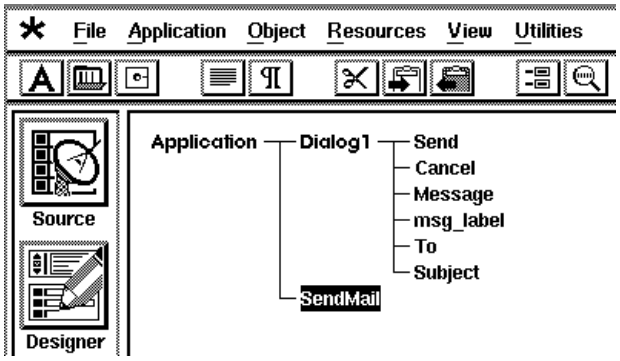
When you run the application the dialog box is displayed.

When you click on Send Message the Send Mail dialog box appears. Set the message attributes and click on OK to send a message.



Advanced MailClass Example

The following is an advanced example of a mail application. The application sends a certified message with a reply request to one person. Replies to the message are sent to MyAssistant, instead of to the message sender. The application contains a MailClass object named SendMail, a dialog box with two push buttons named Send and Cancel, two entry fields named To and Subject, and an edit box named Message.



Send's source code contains the following lines in the clicked_event to send the mail message, initializing the recipient, subject, and message body to the information typed into the dialog box.

```
@@@ OBJECTS
set clicked_event
        var object msg, recip,subj,text
        msg    = this.application@.child@("SendMail")
        recip  = this.sibling@("To")
        subj   = this.sibling@("Subject")
        text   = this.sibling@("Message")
        msg.to_recips@ = recip.value@
        msg.subject@ = subj.value@
        msg.body@ = text.value@
        msg.is_certified@ = TRUE
        msg.is_reply_requested@ = TRUE
        msg.reply_text@ = "Send a reply ASAP"
        msg.alternate_reply_recipient@ = "MyAssistant"
        msg.send_mail@
        this.application@.quit@
endset
```

Cancel's source code contains the following lines in the clicked_event to exit the application.

```
@@@ OBJECTS
set clicked_event
        this.application@.quit@
endset
```

When you run the application the dialog box is displayed.

When you click on Send Message the message is sent and the application exits. The interactive status of the SendMail object is FALSE by default. The Send Mail dialog box is displayed only if a recipient or subject is not supplied for the message.

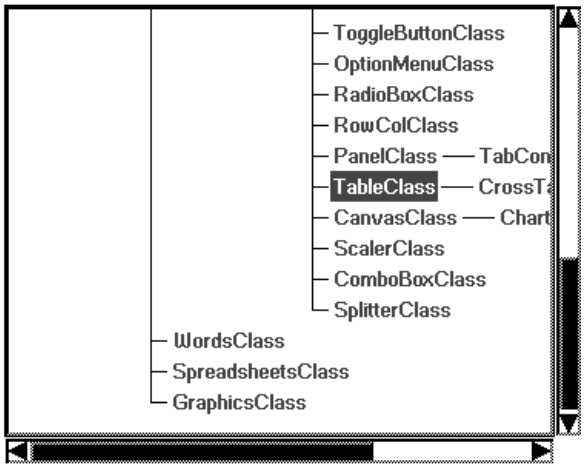
12 Using TableClass Methods

A table displays information in columns and rows. A table can have horizontal and vertical scroll bars so that you can move through the table information. Use the TableClass methods to set and retrieve table information. This chapter covers the following topics:

- TableClass overview
- Creating a TableClass object
- Using a Table

TableClass Overview

Use the TableClass methods to manipulate table data in a dialog box. A table allows you to compare multiple sets of data at the same time. You can connect a table to a data set or Real Time data feed to display their information, or you can place data directly in the table from other sources.



Creating a TableClass Object

The TableClass object exists in a dialog box. To create a table in a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Table** in the Dialog Box Editor window.
7. Click in the dialog box to place the table.
8. Resize the table and set the table attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a table. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for information on creating and editing dialog boxes.

Using a Table

You can display database, real time data feed, and other types of information in a table. See "Connecting a Table" in Chapter 5 in the

Applix Builder User's Guide, "Using the Connector Tool", for information on connecting data sets and real time data feeds to a table.

The TableClass methods generally fall into the following categories:

- Table attributes and information
- Table editing and navigation

The TableClass also contains several events you can program for table actions.

Setting Table Attributes and Information

The TableClass methods that set table attributes and information include methods to set:

- column headers
- row markers
- data set or real time fields
- table data

Setting Column Headers

Use column headers to identify the column information displayed in the table. Use the heading_info@ method to set the column headings and column width. Use the title_ methods, such as title_color@ and title_font_attrs@, to set the heading text attributes.

Setting Row Markers

The TableClass has several methods to control the appearance and content of row markers. Use row markers to distinguish row information in the table, especially in tables containing many rows. You can set row markers at any time during application execution. For example, you can set the row markers for edited or selected rows in the table. To use row markers, you must first set the row_markers_is_suppressed@ method to FALSE, then you must set the marker_width@, which is in pixels, to a width greater than 0.

After you set the marker width, you can display numbers, strings, or bitmaps in the markers. Set the row_numbers_is_enabled@ method to FALSE to use bitmaps or strings in the row markers. The marker_pixmaps@ and marker_strings@ methods take the starting row as the first argument. Use these methods in the table's update_event. If you set the string or bitmap information in the initialize_event, before the table is displayed, the strings or pixmaps may not appear in the markers. The marker_pixmaps@ method takes a two-dimensional array of bitmap names as the second argument, while the marker strings@ method takes a one-dimensional array of strings as the second argument. Set the row_numbers_is_enabled@ method to TRUE to use row numbers in the row markers. Row numbering starts at 1, although row information is a 0-based array.

The corner_pixmaps@ method sets the bitmaps displayed in the upper left corner of the table, where the row markers and column headers intersect.

Connect Data Sets and Data Feeds

You can connect data sets and real time data feeds to a table with the Connector. You can also use the ControlClass and TableClass methods to define a data source for the table. To define a data source using methods, you need to set the data source type with the ControlClass method data_source_type@. After you define the data source type as a data set or a real time data feed you can set the data source information.

If the table's data source is a data set, use the ControlClass method data_set@ to assign the data set in your application to the table. After you assign the data set, use the TableClass method data_set_field_name_list@ to set the fields that appear in the table. The column names appear by default as the column headers. Use the heading_info@ method after you set the field names to set the column width and header text. The quantity of columns in the table depends on the information returned by the data set query.

If the table's data source is a real time data feed, use the Connector to define the initial gateway, record, and field information. Each record displayed in a table must contain the same fields, a table cannot display different fields for each record. Use the TableClass methods rt_record_list@ and rt_field_list@ to change the records and fields from the settings in the connector during the course of application execution. If you do not use the Connector, the table information can only be updated in with the data_update_event of each RealtimeRecordClass object associated with the table. Using the data_update_event to set table data for each record can create considerable operational overhead during application execution.

Setting Table Data

Use the table_data@ method to set table information from sources other than a data set or real time data feed. The data passed to the table_data@ method must be a two-dimensional array. The first dimension determines the row, the second dimension defines the column in each row. The set method value@ sets table information if table data is NULL. The value@ method also takes a two-dimensional array of table information as an argument. After you initially set table information with the value@ method use the table_data@ method to change table information. For example, the following initialize_event sets three rows of table data, each row contains two columns of information.

```
set initialize_event
        var head, data
        /* Set heading info for 2 columns  */
        head[0,0] = "Head1"
        head[0,1] = 100
        head[1,0] = "Head2"
        head[1,1] = 100
        /* Set 3 rows of table data  */
        data[0,0] = "Row 0, Cell 0"
        data[0,1] = "Row 0, Cell 1"
        data[1,0] = "Row 1, Cell 0"
        data[1,1] = "Row 1, Cell 1"
        data[2,0] = "Row 2, Cell 0"
        data[2,1] = "Row 2, Cell 1"
        this.heading_info@ = head
        this.value@ = data
endset
```

Table Editing and Navigation

After you set table attributes and information, you can use the editing and navigation methods to:

- Set table editing, column resizing, and row selecting states
- Set one row of table information
- Insert text in a cell
- Set selected rows
- Go to a specific cell
- Set the top table row

Use the set method is_editable@ to set the table editing state. The table, by default, is in a read-only state. Set is_editable@ to TRUE to allow table editing. The method cell_editing_is_allowed@ works the same as is_editable@.

Use the set method column_resizing_is_allowed@ to set the column resizing state in the table. Column resizing is allowed by default, set column_resizing_is_allowed@ to FALSE to disable column resizing.

Use the set method is_multi_select@ to allow multiple row selections in the table. The table, by default, only allows one row to be selected at one time. Set is_multi_select@ to TRUE to allow multiple row selections in the table using CTRL-Click (press the CTRL key while clicking the left mouse button) or a click and drag across rows.

The editing methods allow you to set information for a given row or cell. The set method one_row@ sets one row of information in the table. Use the method to add or change information in a table that already contains data. The set method insert_text@ inserts a text string in the cell containing the text cursor.

Use the set method selections@ to set the selected rows in the table. The method takes an array of row numbers as an argument; table row numbers are 0-based. Use the method goto_cell@ to place the cursor in a table cell, at a specified character position. For example, you can use the method in a search algorithm to place the cursor in a cell matching search conditions. Use the set method top_row@ to set the top row displayed in the table.

Using TableClass Events

You can program the TableClass events to perform edit verification, updates, or any actions necessary for the proper functioning of your application. The sequence of event calls is discussed in Appendix A, "Sequence of Events".

The following events are called in the TableClass:

button3_menu_state_event
error_event
cell_changed_event
cell_focus_in_event
cell_focus_out_event
column_resize_event
double_click_event
initialize_event
request_data_event
resize_event
selection_changed_event
terminate_event
time_out_event
typing_event
update_event

The button3_menu_state_event is called before a popup menu is displayed. A popup menu is a free-floating menu associated with a table. Popup menu information is set with the button3_menu_info@ method. A popup menu can be activated when the mouse pointer is in the control area.

The request_data_event is called when a vertical table scroll occurs and the model_is_data_request@ method is set to TRUE. Use the event to place data in the rows exposed in the table viewing area. Use a table in data request mode to minimize the amount of data you need to load into the table. If you do not place data in the exposed rows they will remain blank. The table does not retain the data in the table, when a row containing data scrolls out of the viewing area its contents are cleared.

The selection_changed_event is called when a row is selected with a mouse button click. This includes a single row selection, or the addition of a row to multiple row selections. The event is not called when a selection is made with the set method selections@.

The cell_focus_in_event is called when you click in a table cell for the first time. The cell_focus_out_event is called when you click out of the current cell into a different cell. The cell_changed_event is called when the change the contents of the current cell and click in a different cell.

Examples

TableClass examples are available in the table directory. Choose Tools → Sample Applications to access the table directory or other sample application directories with the Directory Displayer dialog box. The examples provide an introductory usage of methods and events in the TableClass.

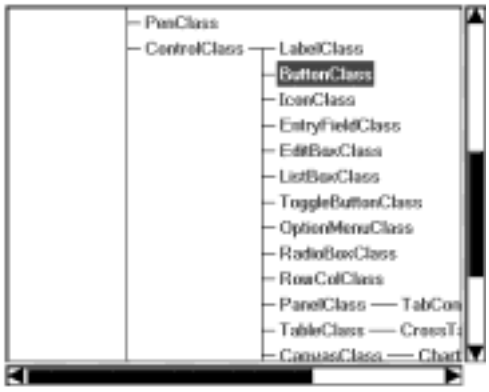
13 Using ButtonClass Methods

A push button performs a programmed action in an application when a users clicks on it. The actions can include closing the dialog box, opening a new dialog box, or launching a new application. This chapter covers the following topics:

- ButtonClass overview
- Methods and events
- Creating a ButtonClass object
- Using a ButtonClass object

ButtonClass Overview

Use a ButtonClass object to initiate actions in an application in response to a mouse click. A ButtonClass object appears as a push button in a dialog box.



Methods and Events

The following ButtonClass methods and events are discussed in this chapter:

is_default@	value@
height@	width@
parent@	clicked_event
title@	

Creating a ButtonClass Object

A ButtonClass object is added as a control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. When you create and edit a dialog box, a Dismiss push button is inserted by default in the dialog box. The Dismiss push button is programmed to dismiss its parent dialog box when it is clicked. You can remove the Dismiss push button or change its functionality to suit your application's needs. To create a push button in a dialog box:


Designer icon

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.

4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Push Button** in the Dialog Box Editor window.
7. Click in the dialog box to place the push button.
8. Resize the push button and set the push button attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a push button. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using a ButtonClass Object

The primary purpose of a push button is start or cancel operations in an application. You need to program a push button clicked_event to perform actions in response to a push button click. For example, the following actions are defined in the Dismiss push button clicked_event:

```
set clicked_event
        this.parent@.close@
endset
```

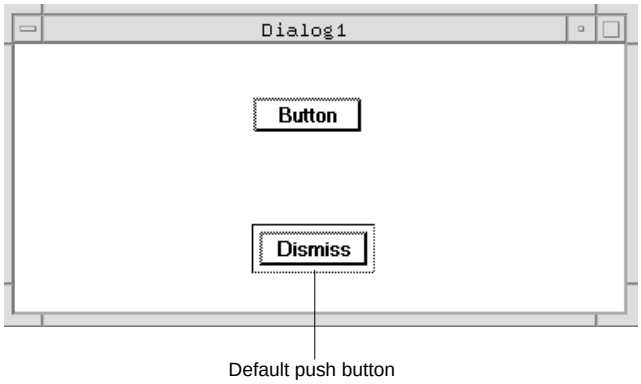
The push button uses the parent@ method to get a handle to its parent, the dialog box containing the push button. The push button uses the handle to the dialog box to call the DialogBoxClass close@ method, which closes the dialog box.

In addition to the clicked_event, you can set the attributes of a push button with the ButtonClass methods. The set method title@, inherited from the ControlClass, sets the title string displayed within the push button. For example, the following clicked_event is programmed to change the title of the push button to reflect how many times you clicked on the button. The objvar num_pushed is set to zero in the initialize_event, then incremented by one in the clicked_event.

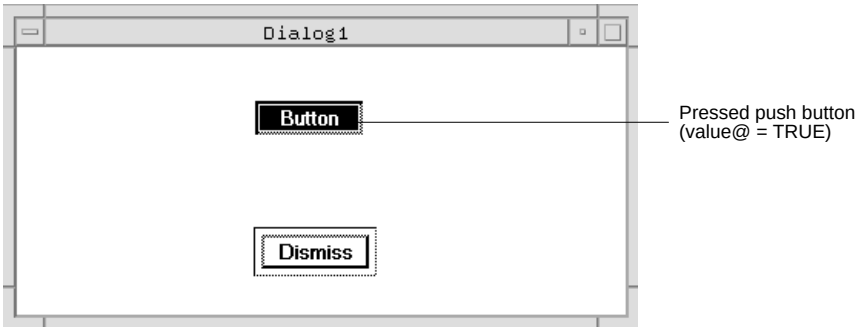
```
objvar num_pushed
set initialize_event
        this.title@ = "Button Not Pushed"
        num_pushed = 0
endset
set clicked_event
        num_pushed = num_pushed + 1
        this.title@. = "Button Pushed " ++ num_pushed
        if (num_pushed = 1)
                this.title@. =this.title@ ++ " time"
        else
                this.title@. =this.title@ ++ " times"
endset
```

The set method is_default@ sets the default state of a push button. A dialog box can have only one default push button. If the is_default@

method is set to TRUE, the push button is selected when you press ENTER or RETURN. A default push button is displayed with an outlining rectangle. You can assign the default status to a push button in an initialize_event, or during the application execution to reflect the state of the application or the other controls in the dialog box.



The set method value@ sets the pressed appearance of a button in a dialog box. When you click on a push button, the push button usually appears pressed during the click, then returns to its normal appearance after the click. You can use the value@ method to use a push button to reflect the state of a program, or use several buttons as a radio button grouped managed by your application, with the value@ set for the selected button. Set the value@ method to TRUE to make a push button pressed.



Use the set methods height@ and width@ to set the height and width, in pixels, of a push button. The width and height define the minimum height and width of a push button. A push button will expand to display the full title string, even if the string length or font size make the string exceed the width and height dimensions.

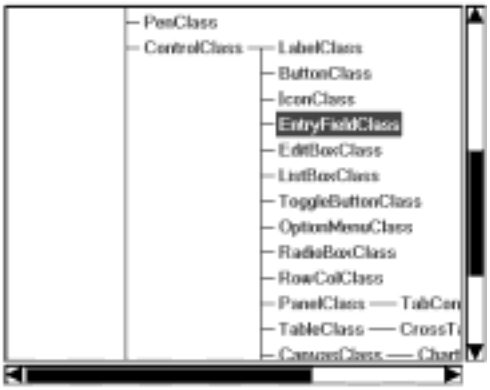
14 Using EntryFieldClass Methods

An entry box is a dialog box control that allows you to enter a single line of text in an application. This chapter covers the following topics:

- EntryFieldClass overview
- Methods and events
- Creating an EntryFieldClass object
- Using a EntryFieldClass object

EntryFieldClass Overview

Use an EntryFieldClass object for single-line text entry in an application. Use an entry box to input a single line of text into your application, such as a user name or password.



Help for EntryFieldClass methods is available through the Class Browser dialog box. Choose View → Classes to use the Class Browser dialog box.

Methods and Events

The following EntryFieldClass methods and events are discussed in this chapter:

button3_menu_info@	title_position@
cursor_pos@	valid_chars@
is_numeric@	value@
is_optional@	button3_menu_state_event
is_password@	changed_event
is_selected@	focus_in_event
is_trimmed@	focus_out_event
max_chars_length@	typing_event

Creating an EntryFieldClass Object

An EntryFieldClass object is added as an entry box control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. To create an entry box in a dialog box:


Designer icon

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.

3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Entry Box** in the Dialog Box Editor window.
7. Click in the dialog box to place the entry box.
8. Resize the entry box and set the entry box attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with an entry box. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using an EntryFieldClass Object

Use an entry box to input a single line of text into your application. In addition to setting entry box attributes through the Dialog Box Editor window, you can use EntryFieldClass methods and events to set or change:

- the type of information allowed in an entry box
- how information is displayed in an entry box
- the actions that occur in an entry box

Setting EntryFieldClass Information

Each EntryFieldClass object has a set of attributes you can set to control the type of information allowed in an entry box.

Use the set method is_optional@ to determine whether text is required in the entry box. If you set is_optional@ to TRUE, text is optional in the entry box. If you set is_optional@ to FALSE, then text is required in the entry box before the dialog box can be dismissed. You can make text mandatory to ensure that your application receives the data it needs to continue processing properly.

Use the set method is_numeric@ to limit entry box input to numeric input. Set is_numeric@ to TRUE to allow only numeric input in the entry box. For example, if your application is performing a calculation on the value in the entry box, this method can restrict input to numeric characters, preventing errors caused by using non-numeric input.

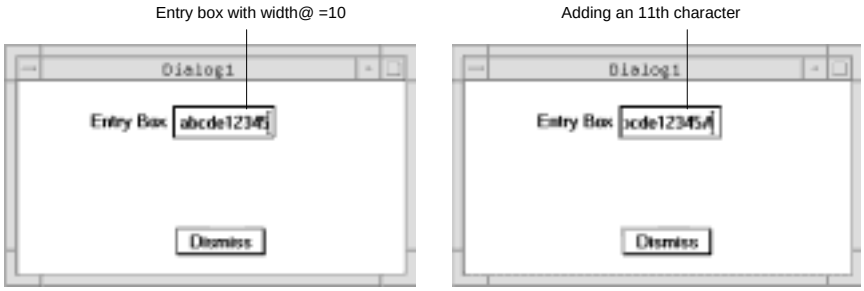
Another method you can use to limit character input is the set method valid_chars@. The method takes a string as an argument. The string contains the characters accepted by the entry box. For example, the following initialize_event for an EntryFieldClass object uses the valid_chars@ method.

```
set initialize_event
        this.valid_chars@ = "ABC123"
endset
```

The entry box only accepts the characters in the string passed to valid_chars@. If you type x or 4, these characters are not accepted, the

terminal will beep for each invalid character, but the cursor position will not change.

You can use the set method max_chars_length@ to limit the number of characters allowed in the entry box. The method takes an integer as an argument. The integer is the maximum number of character allowed in the entry box. The maximum number of characters is not related to the entry box width, which is the number of characters visible in the entry box. When you type a string greater than the width, and the max_chars_length@ is set greater than the entry box width@, the string scrolls left in the entry box and the last character typed appears in the entry box. For example, the following dialog box has width@ = 10 and max_chars_length@ = 20. The first ten characters of a string appear in the entry box, but the 11th character cause the string to scroll left.



Use the set method value@ to load a pre-defined string or value in an entry box. You can use the method to present default values in an application, instead of typing values for each entry box. For example, the following initialize_event for an EntryFieldClass object uses the value@ method to initialize the value in the entry box to Start.

```
set initialize_event
        this.value@ = "Start"
endset
```

When you run the application, the entry box appears with the initial value Start selected in the entry box.

Setting EntryFieldClass Display Information

In addition to the methods that set the color of the entry box, the title, and the text that appears within the entry box, the EntryFieldClass has methods you can use to control how information is displayed in the entry box. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the set method is_password@ to use the entry box for password entry. If you set the method to TRUE, each character typed into the entry box is displayed as an asterisk. The actual string typed into the entry box can be retrieved with the get method value@.

Use the set method is_selected@ to select or deselect the entry box contents. Set the method to TRUE to select the entry box contents, or to FALSE to deselect the entry box contents. Use this method after the entry box is initialized and displayed.

Use the set method cursor_pos@ to set the cursor position in the entry box. Cursor positions begin with 1, the left most position in the entry box. If you set the method to 0 the cursor is not displayed in the entry box. Use this method after the entry box is initialized and displayed. For example, the following initialize_event and update_event for an

EntryFieldClass object set the initial value of an object, then set the cursor position to 2, before the second character in the string.

```
set initialize_event
        this.value@ = "Start"
endset
set update_event
        this.is_selected@ = FALSE
        this.cursor_pos@ = 2
endset
```

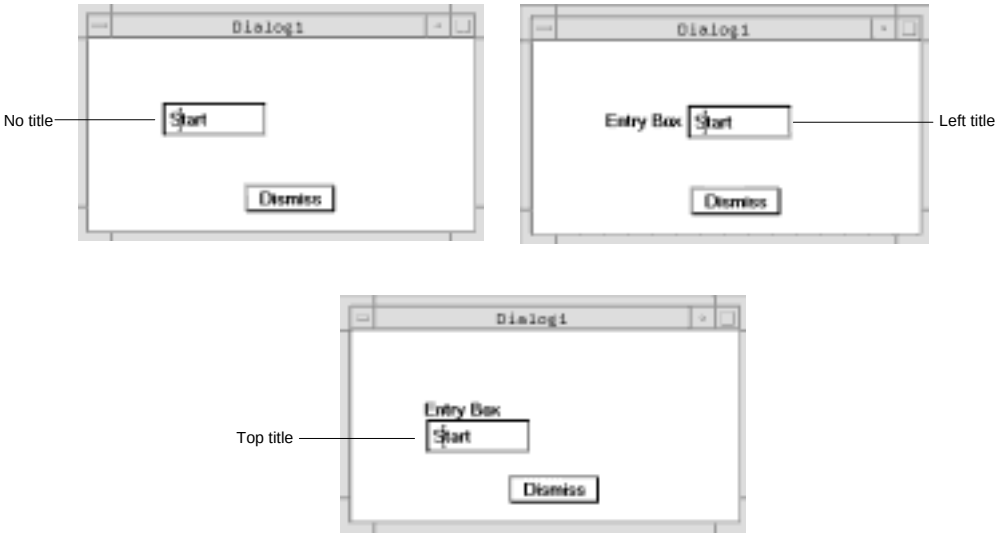
When you run the application, the entry box appears with the initial value Start in the entry box and the cursor set before the second character.

The set method title_position@ affects the entry box title, not the contents of an entry box. All controls in a dialog box have a title, although some controls do not display their title. You can choose the display position of aentry box title. An entry box can

- display a title to the left of the entry box
- display a title above the entry
- display no title

You can set the title_position@ method to one of three valid values. Pass the value -1 to the method to display no title for the entry box. Pass the value 0 to the method to display the title to the left of the

entry box. Pass the value 1 to the method to display the title above the entry box.



Programming EntryFieldClass Actions

Actions in an EntryFieldClass object generate events. For example, typing in an entry box, tabbing to a different control in the dialog box, or pressing the right mouse button are actions that generate events. You must program the defined events for an EntryFieldClass object to respond to the actions and handle them accordingly. The events of particular importance to an EntryFieldClass object are:

- button3_menu_state_event
- changed_event
- focus_in_event

- focus_out_event
- typing_event

See Appendix A, "Sequence of Events", for information about all the events available in an EntryFieldClass.

Focus_in_event and focus_out_event

A focus_in_event is called when the entry box receives focus in the dialog box, either by clicking in the entry box or tabbing to the entry box from a different control. You can use a focus_in_event to set the cursor position or otherwise prepare the entry box for input. For example, the following focus_in_event sets the value and cursor position when an entry box receives focus.

```
set focus_in_event
        this.value@ = "Focus in"
        this.is_selected@ = FALSE
        this.cursor_pos@ = 3
endset
```

A focus_out_event is called when the entry box loses focus, either by clicking on a differentr dialog box control or tabbing out of the entry box to the next dialog box control that can receive focus. You can use a focus_out_event to perform cleanup of of an entry box.

changed_event

A changed_event is called when an entry box value changes and the entry box loses focus. A changed_event is called before a focus_out_event. A changed_event is passed the new value in the entry box as an argument. You can use a changed_event to validate a

changed value in the entry box before passing the value along to your application. For example, the following changed_event compares the entry box value against the string Start. If the current value does not match Start, the entry box value is set to Start.

```
set changed_event(val)
        if val <> "Start"
                this.value@ = "Start"
endset
```

You could extend this example to compare the value against an array of strings, such as checking for a valid user ID.

typing_event

A typing_event is called when a character is typed in the entry box, or when you type BACKSPACE or DELETE when the entry box has focus. You can use the typing_event for entry box value validation, such as proper user ID format. For example, in the following typing_event for an EntryFieldClass object, if the first character of the string is numeric, an info message is posted stating the restriction, then the entry box value is cleared.

```
set typing_event
        var val
        val = SUBSTRING@(this.value@,1,1)
        IF IS_NUMERIC_STRING@(val)
        {
        INFO_MESSAGE@("U ID Cannot begin with a
                        number!")
                this.value@ = NULL
        }
endset
```

Use the set method is_trimmed@ to remove leading and trailing spaces from values returned by the entry box. Set the method to TRUE to remove the leading and trailing spaces from the value. Set the method to FALSE to return the value with all spaces, as it appears in the entry box. The is_trimmed@ setting only affects the values returned by the get method value@ and the value passed to the changed_event.

button3_menu_state_event

You can program a popup menu for an entry box that appears when you press the right mouse button in the clickable area for the entry box. You can use a popup menu to provide quick access to macros or methods. Use the set method button3_menu_info@ to set the popup menu information, and the button3_menu_state_event to set the state of the menu items before the popup menu is displayed.

The button3_menu_info@ method takes an array of arguments in rminfo@ format. The format is defined in the dialog_.am file, located in the *install_dir*/axdata/elf directory. Each menu item supplies the name displayed in the menu, the method called when the menu item is selected, and the active state of the menu item.

The button3_menu_state_event is called before the popup menu is displayed, to set the state of each menu item. This event is a get event, which must return a value. The method name of each menu item is passed to the button3_menu_state_event. The event is called once for each menu item. You program the event to check the function name, and to return a value to set the menu item state. The event should return TRUE to make the menu item active. The event should return FALSE to gray out the menu item and make it inactive. The event should return NULL to keep the current state of the menu item.

In the following example, an initialize_event defines a popup menu with the button3_menu_info@ method. The menu contains three menu items, calling user-defined methods. A button3_menu_state_event is defined, which grays the menu item

containing the set_val user-defined method, and leaves all other menu items unchanged.

```
#include "dialog_.am"
set initialize_event
        var format arrayof rminfo@ popup

        popup[0].name = "Clear"
        popup[0].macro_name = "clear"
        popup[0].active = TRUE
        popup[1].name = "Grayed"
        popup[1].macro_name = "set_val"
        popup[1].active = TRUE
        popup[2].name = "Exit Dbox"
        popup[2].macro_name = "quit"
        popup[2].active = TRUE

        this.button3_menu_info@ = popup
        this.is_trimmed@ = TRUE
endset

get button3_menu_state_event(func)
        case of func
        case "set_val"
                return(FALSE)
        default
                return(NULL)
        endcase
endget
```

```
set clear
        this.value@ = NULL
endset

set quit
        this.parent@.close@
endset

set set_val
        this.value@ = "Default"
endset
```

When you run the application and click the right mouse button with the cursor in the entry box area, a popup menu appears. The Grayed menu item, which calls the set_val user-defined method, appears grayed-out. Even though the menu item is defined as active in the initialize_event, the button3_menu_state_event makes the menu item inactive. You can use the button3_menu_state_event to dynamically activate and deactivate menu items, the previous example was a static use of the event.

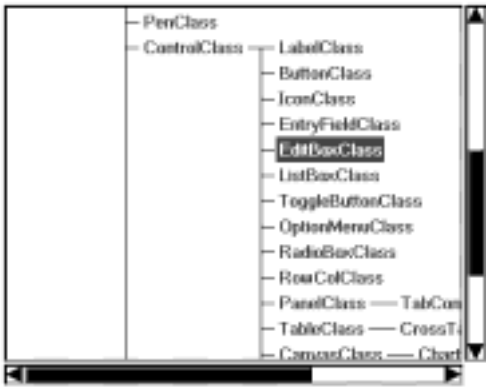
15 Using EditBoxClass Methods

An edit box is a multiple line text input and editing control. This chapter covers the following topics:

- EditBoxClass overview
- Methods and events
- Creating an EditBoxClass object
- Using a EditBoxClass object

EditBoxClass Overview

Use an EditBoxClass object for multiple line text entry and editing in your application. You can program an edit box to input to text to your application, or to display text from your application, such as comments or operating instructions.



Methods and Events

The following EditBoxClass methods and events are discussed in this chapter:

button3_menu_info@	text_cursor@
copy_to_clipboard@	value@
cursor_line_number@	word_at_cursor@
cut_to_clipboard@	button3_menu_state_event
is_mono_space@	changed_event
is_read_only@	focus_in_event
paste_from_clipboard@	focus_out_event
selected_word@	typing_event
selection@	

Creating an EditBoxClass Object


Designer icon

An EditBoxClass object is added as an edit box control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. To create an entry box in a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.

2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Edit Box** in the Dialog Box Editor window.
7. Click in the dialog box to place the edit box.
8. Resize the entry box and set the entry box attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with an entry box. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using an EditBoxClass Object

Use an edit box for multiple line text entry and editing in your application. In addition to setting edit box attributes through the Dialog Box Editor window, you can use EditBoxClass methods and events to set or change:

- how information is displayed in an edit box
- the selection and usage of selected information
- the actions that occur in an edit box

Setting EditBoxClass Display Information

In addition to the methods that set the color and size of the edit box, the EditBoxClass has methods you can use to control how information is displayed in the edit box. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the set method value@ to load a pre-defined set of strings in an edit box. You can use the method to present default strings in an application, instead of typing strings into the edit box. For example, the following initialize_event for an EditBoxClass object uses the value@ method to initialize the value in the edit box.

```
set initialize_event
    this.value@ = {"First line in the edit box","Second line in
                   the edit box", "Third line in the edit box"}
endset
```

When you run the application, the edit box appears with the initial set of strings.

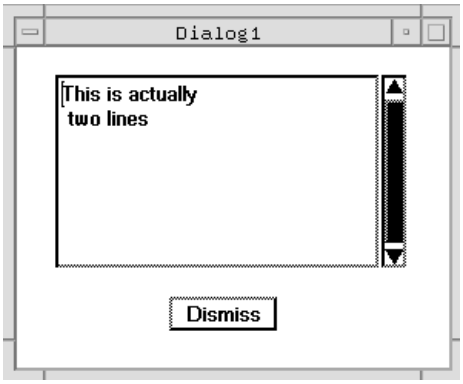
Use the set method is_read_only@ to set the read-only status of the edit box. Set the method to TRUE to make the edit box read-only. Set the method to FALSE to allow changes to the edit box contents. The edit box is editable by default.

Use the set method is_mono_space@ to set the text font spacing. Set the method to TRUE to use the default monospace font in the edit box. Set the method to FALSE to use the standard default font. If you set a text font for the edit box, the is_mono_space@ settings have no effect on the font.

Selecting and Using EditBoxClass Information

You can use EditBoxClass methods to select and manipulate edit box strings. Use the selection and cursor methods to choose the information, then use the clipboard methods to move or change the information.

A line is a string of text terminated by a carriage return. Text automatically wraps in an edit box until you press RETURN or ENTER, or place the carriage return character sequence (\n) in a string when you set the value@. For example, the string This is actually \n two lines has the carriage return character sequence in the middle of the string. The string appears in the edit box as:



Use the get method cursor_line_number@ to get the line number of the current cursor position in the edit box. Cursor positions begin at line 0.

Use the set method text_cursor@ to set the cursor position in the edit box. The method takes the line number and character position as arguments. Character positions begin at 0, before the first character in the line. You can use this method with the cursor_line_number@ method to set the cursor position in the edit box.

You can select text in an edit box by dragging the mouse over the text. You can also use the set method selection@ to select text programmatically. The method takes a four-element array as an argument. The argument array contains the following:

array[0]	The beginning line number, 0-based.
array[1]	The beginning character position, 0-based.
array[2]	The end line number, 0-based.
array[3]	The end character position, 0-based.

The related get method returns the selection information in the same array format.

The EditBoxClass contains several methods you can use to get information from the edit box. The get method word_at_cursor@

returns the string at the current cursor position. The method returns the string to the right of the cursor, ignoring leading spaces. If the cursor is in a string, the string is returned.

Use the get method selected_word@ to return the selected words on a single line. The selected words are returned as a single string by the method.

Use the clipboard methods to move or change information in the edit box. The clipboard is an area of memory. You can copy information to the clipboard, where it is available until overwritten by new information. You can use the clipboard methods to make multiple copies of a string in an edit box, or to move strings to a different line in the edit box.

Use the set method copy_to_clipboard@ to copy the selected text to the clipboard. Use the set method cut_to_clipboard@ to copy the selected text to the clipdboard and remove the text from the edit box. When text is placed in the clipboard, use the set method paste_from_clipboard@ to paste the text into the edit box. The method places the text into the edit box at the current cursor position.

Programming EditBoxClass Actions

Actions in an EditBoxClass object generate events. For example, typing in an edit box, tabbing to a different control in the dialog box, or pressing the right mouse button are actions that generate events. You must program the defined events for an EditBoxClass object to respond to the actions and handle them accordingly. The events of particular importance to an EditBoxClass object are:

- button3_menu_state_event
- changed_event
- focus_in_event

- focus_out_event
- typing_event

See Appendix A, "Sequence of Events", for information about all the events available in an EditBoxClass.

focus_in_event and focus_out_event

A focus_in_event is called when the edit box receives focus in the dialog box, either by clicking in the edit box or tabbing to the edit box from a different control. You can use a focus_in_event to set the cursor position or otherwise prepare the edit box for input. For example, the following focus_in_event sets the value and cursor position when an edit box receives focus.

```
set focus_in_event
        this.value@ = {"Focus in","Line 2"}
        this.text_cursor@(1,0)
endset
```

A focus_out_event is called when the edit box loses focus, either by clicking on a different dialog box control or tabbing out of the edit box to the next dialog box control that can receive focus. You can use a focus_out_event to perform cleanup of of an edit box.

changed_event

A changed_event is called when an edit box value changes and the edit box loses focus. A changed_event is called before a focus_out_event. A changed_event is passed the new value in the edit box as an argument. You can use a changed_event to validate a changed value in the edit box before passing the value along to your application. For example, the following changed_event compares the

first edit box value against the string Start. If the current value does not match Start, the edit box value is set to Start.

```
set changed_event(val)
        if val[0] <> "Start"
                val[0] = "Start"
                this.value@ = val
endset
```

You could extend this example to compare all returned strings in the value.

typing_event

A typing_event is called when a character is typed in the edit box, or when you type BACKSPACE or DELETE when the edit box has focus. You can use the typing_event for edit box value validation, such as proper user ID format or character validation. For example, in the following typing_event for an EditBoxClass object, if the first line in the edit box contains the string foo the line is made blank.

```
set typing_event
        var val
        val = this.value@
        IF (STRING_INDEX@(val[0],"foo") <> 0 )
        {
                val[0] = NULL
                this.value@ = val
        }
endset
```

button3_menu_state_event

You can program a popup menu for an edit box that appears when you press the right mouse button in the clickable area for the edit box. You can use a popup menu to provide quick access to macros or methods. Use the set method button3_menu_info@ to set the popup menu information, and the button3_menu_state_event to set the state of the menu items before the popup menu is displayed.

The button3_menu_info@ method takes an array of arguments in rminfo@ format. The format is defined in the dialog_.am file, located in the *install_dir*/axdata/elf directory. Each menu item supplies the name displayed in the menu, the method called when the menu item is selected, and the active state of the menu item.

The button3_menu_state_event is called before the popup menu is displayed, to set the state of each menu item. This event is a get event, which must return a value. The method name of each menu item is passed to the button3_menu_state_event. The event is called once for each menu item. You program the event to check the function name, and to return a value to set the menu item state. The event should return TRUE to make the menu item active. The event should return FALSE to gray out the menu item and make it inactive. The event should return NULL to keep the current state of the menu item.

In the following example, an initialize_event defines a popup menu with the button3_menu_info@ method. The menu contains three menu items, calling user-defined methods. A button3_menu_state_event is defined, which grays the menu item containing the set_val user-defined method, and leaves all other menu items unchanged.

```
#include "dialog_.am"
set initialize_event
        var format arrayof rminfo@ popup

        popup[0].name = "Clear"
```

```
        popup[0].macro_name = "clear"
        popup[0].active = TRUE
        popup[1].name = "Grayed"
        popup[1].macro_name = "set_val"
        popup[1].active = TRUE
        popup[2].name = "Exit Dbox"
        popup[2].macro_name = "quit"
        popup[2].active = TRUE
        this.button3_menu_info@ = popup
endset

get button3_menu_state_event(func)
        case of func
        case "set_val"
                return(FALSE)
        default
                return(NULL)
        endcase
endget

set clear
        this.value@ = NULL
endset

set quit
        this.parent@.close@
endset

set set_val
        this.value@ = {"Default edit box text"}
endset
```

When you run the application and click the right mouse button with the cursor in the edit box area, a popup menu appears. The Grayed menu item, which calls the set_val user-defined method, appears grayed-out. Even though the menu item is defined as active in the initialize_event, the button3_menu_state_event makes the menu item inactive. You can use the button3_menu_state_event to dynamically activate and deactivate menu items, the previous example was a static use of the event.

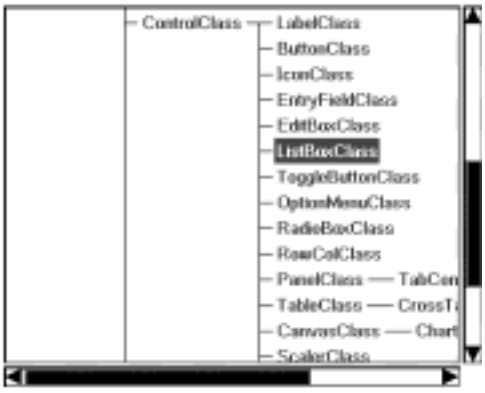
16 Using ListBoxClass Methods

A list box is a multiple line text display and selection control. This chapter covers the following topics:

- ListBoxClass overview
- Methods and events
- Creating a ListBoxClass object
- Using a ListBoxClass object

ListBoxClass Overview

Use a ListBoxClass object for multiple option display and selection in your application. Each list box choice is a separate line in the list box. .



Methods and Events

The following ListBoxClass methods and events are discussed in this chapter:

button3_menu_info@	value_index@
hscroll_is_enabled@	vscroll_is_enabled@
hscroll_length@	vscroll_length@
hscroll_origin@	vscroll_origin@
is_mono_space@	button3_menu_state_event
is_multi_select@	changed_event
is_read_only@	double_click_event
text_strings@	multi_select_event
value@	stroke_select_event

Creating a ListBoxClass Object


Designer icon

A ListBoxClass object is added as a list box control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. To create a list box in a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.

2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → List Box** in the Dialog Box Editor window.
7. Click in the dialog box to place the edit box.
8. Resize the list box and set the list box attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a list box. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using a ListBoxClass Object

Use a list box for multiple option display and selection in your application. In addition to setting list box attributes through the Dialog Box Editor window, you can use ListBoxClass methods and events to set or change:

- information display and access in a list box
- the actions that occur in a list box

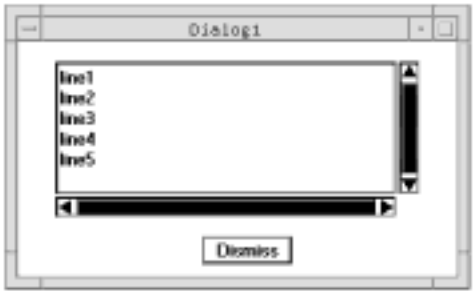
ListBoxClass Information Display and Access

In addition to the methods that set the color and size of the list box, you can use ListBoxClass methods to control how information is displayed in the edit box. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the set method text_strings@ to load the list box information. For example, the following initialize_event for a ListBoxClass object uses the text_strings@ method to initialize the value in the edit box.

```
set initialize_event
        this.text_strings@ =
                {"line1","line2","line3","line4","line5"}
endset
```

When you run the application, the list box appears with the initial set of strings.



A list box can have horizontal and vertical scroll bars. Users of your application can use scroll bars to access list box information that is not currently or completely visible in the viewable area. When you add a list box to a dialog box, the horizontal and vertical scroll bars are enabled by default. Use the set methods hscroll_is_enabled@ and vscroll_is_enabled@ to enable or disable horizontal or vertical scroll bars. The methods take a Boolean argument. Pass the method TRUE to enable a scroll bar, or pass the method FALSE to disable a scroll bar.

Use the set method is_mono_space@ to set the text font spacing. Set the method to TRUE to use the default monospace font in the list box. Set the method to FALSE to use the standard default font.

You can use ListBoxClass methods to set the selection access of information in the list box. You can allow single selections, multiple selections, or no selections in the list box.

The list box allows single item selections by default. Use the set methods is_multi_select@ and is_read_only@ to change the default selection characteristics. Set the is_multi_select@ method to TRUE to allow multiple selections with CTRL-click actions. Set the method to FALSE to allow only single selections in the list box. Set the is_read_only@ method to TRUE to make the list box read-only, allowing no selections. Set the method to FALSE to allow selections in the list box.

You can set or determine the selections in a list box using the set and get methods value@ and value_index@. The arguments and returned

values of the methods depend on the is_multi_select@ setting. If is_multi_select@ is set to TRUE the arguments and returned values are arrays. If is_multi_select@ is set to FALSE the arguments and returned values are individual values, not arrays.

The set method value@ sets selections in the list box using text strings that match values in the list box. The get method value@ returns the selection as a string or array of strings. Use these methods if you want to use or set the selected values directly in your application, instead of converting array indices into values.

The set method value_index@ sets selections in the list box using the index of the list box item. The get method value_index@ returns the indices of selections in the list box. The indices used by the set and get methods are against the 0-based array of values in the list box, which you can get using the get method text_strings@.

Programming ListBoxClass Actions

Actions in a ListBoxClass object generate events. For example, changing a selection, double-clicking on a value, or pressing the right mouse button are actions that generate events. You must program the defined events for an ListBoxClass object to respond to the actions and handle them accordingly. The events of particular importance to an ListBoxClass object are:

- button3_menu_state_event
- changed_event
- double_click_event
- multi_select_event
- stroke_select_event

See Appendix A, "Sequence of Events", for information about all the events available in the ListBoxClass.

changed_event and multi_select_event

A changed_event is called when a list box selection changes. The event is passed the index value of the new choice as an argument. You can use this event to monitor which items get selected in your application. The validity of your list box selection may depend upon the state of other controls in the dialog box, as demonstrated by the following example.

```
set changed_event(val)
        var strings
        var object entry
        entry = this.sibling@("Entry Box")
        strings = this.text_strings@
        IF (strings[val] = entry.value@){
        ...
        '...code for when the selected value = entry box value
        ...}
endset
```

The multi_select_event is called when multiple selections occur in the list box. Set the is_mult_select@ method to TRUE to allow multiple list box selections. The changed_event is called for the first selection, the multi_select_event is called for subsequent CTRL-click selections. The event is also called when an item is deselected from a multiple selection. The changed_event is called if deselecting a list box item leaves only one item selected. The multi_select_event receives an array of item indices as an argument, in the selection order of the items.

stroke_select_event

The stroke_select_event is called when multiple items are selected with a mouse stroke. The event is called when the mouse button is released. A SHIFT-Click (pressing SHIFT and a left mouse button click) key sequence generates a stroke_select_event, selecting list box items from the selected line to the line at the mouse cursor position. The event receives an array of item indices as an argument.

double_click_event

The double_click_event is called when a double-click on a list box item occurs. The changed_event is called first, then the double_click_event is called.

button3_menu_state_event

You can program a popup menu for a list box that appears when you press the right mouse button in the clickable area for the list box. You can use a popup menu to provide quick access to macros or methods. Use the set method button3_menu_info@ to set the popup menu information, and the button3_menu_state_event to set the state of the menu items before the popup menu is displayed.

The button3_menu_info@ method takes an array of arguments in rminfo@ format. The format is defined in the dialog_.am file, located in the *install_dir*/axdata/elf directory. Each menu item supplies the name displayed in the menu, the method called when the menu item is selected, and the active state of the menu item.

The button3_menu_state_event is called before the popup menu is displayed, to set the state of each menu item. This event is a get event,

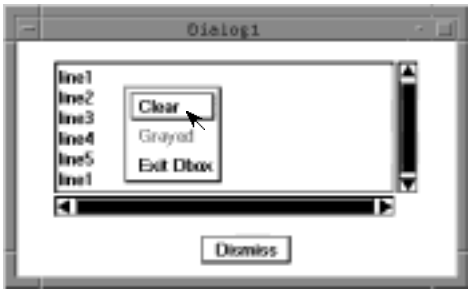
which must return a value. The method name of each menu item is passed to the button3_menu_state_event. The event is called once for each menu item. You program the event to check the function name, and to return a value to set the menu item state. The event should return TRUE to make the menu item active. The event should return FALSE to gray out the menu item and make it inactive. The event should return NULL to keep the current state of the menu item.

In the following example, an initialize_event defines a popup menu with the button3_menu_info@ method. The menu contains three menu items, calling user-defined methods. A button3_menu_state_event is defined, which grays the menu item containing the set_val user-defined method, and leaves all other menu items unchanged.

```
#include "dialog_.am"
set initialize_event
        var format arrayof rminfo@ popup
        this.text_strings@ =
                {"line1","line2","line3","line4","line5","line1"}

        popup[0].name = "Clear"
        popup[0].macro_name = "clear"
        popup[0].active = TRUE
        popup[1].name = "Grayed"
        popup[1].macro_name = "set_val"
        popup[1].active = TRUE
        popup[2].name = "Exit Dbox"
        popup[2].macro_name = "quit"
        popup[2].active = TRUE

        this.button3_menu_info@ = popup
endset
```

```
get button3_menu_state_event(func)
        case of func
        case "set_val"
                return(FALSE)
        default
                return(NULL)
        endcase
endget

set clear
        this.value@ = NULL
endset

set quit
        this.parent@.close@
endset

set set_val
        this.value_index@ = 0
endset
```

When you run the application and click the right mouse button with the cursor in the list box area, a popup menu appears. The Grayed menu item, which calls the set_val user-defined method, appears grayed-out. Even though the menu item is defined as active in the initialize_event, the button3_menu_state_event makes the menu item inactive. You can use the button3_menu_state_event to dynamically activate and deactivate menu items, the previous example was a static use of the event.


Dialog1
line1
line2
line3
line4
line5
line1
Clear
Grayed
Exit Dbox
Dismiss

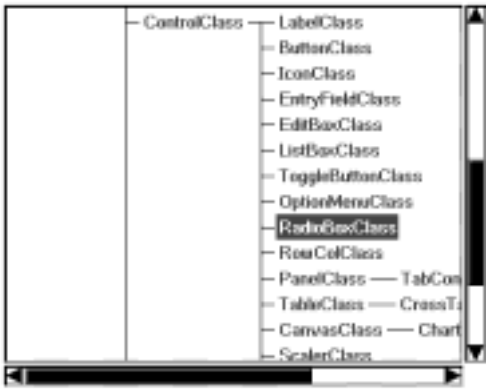
17 Using RadioBoxClass Methods

A radio button group is an exclusive choice control. This chapter covers the following topics:

- RadioBoxClass overview
- Methods and events
- Creating a RadioBoxClass object
- Using a RadioBoxClass object

RadioBoxClass Overview

Use a RadioBoxClass object to present a group of items, from which a user can only choose one item at a time.



Methods and Events

The following RadioBoxClass methods and events are discussed in this chapter:

text_strings@	value_index@
value@	changed_event

Creating a RadioBoxClass Object


Designer icon

A RadioBoxClass object is added as a radio button group in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. To create a radio button group in a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes radio area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Radio Button Group** in the Dialog Box Editor window.

7. Click in the dialog box to place the edit box.
8. Resize the radio button group and set the radio button group attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a radio button group. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using a RadioBoxClass Object

Use a radio button group for exclusive selection from a small group of items. For example, you can have choices that stop or start a process, a user could choose only to start or stop, not both. In addition to setting radio button group attributes through the Dialog Box Editor window, you can use RadioBoxClass methods and events to set or change:

- information display in a radio button group
- the actions that occur in a radio button group

RadioBoxClass Information Display

In addition to the methods that set the color and size of the radio button group, you can use RadioBoxClass methods to control the information displayed as a radio button group. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the set method text_strings@ to set the radio button group items. For example, the following initialize_event for a RadioBoxClass object uses the text_strings@ method to set the radio button group items.

```
set initialize_event
        this.text_strings@ =
                {"Choice1","Choice2","Choice3"}
endset
```

You can set or determine the item selections for a radio button group using the set and get methods value@ and value_index@. The set method value@ sets the selected radio button to the passed string. If the passed string does not match any of the choices, no radio button in the group is selected.

The set method value_index@ sets the selected radio button to the passed index. The index position of radio buttons in the group is 0-based.

The get methods value@ and value_index@ return the current selection in the radio button as, respectively, a string or integer.

Programming RadioBoxClass Actions

Actions in a RadioBoxClass object generate events. When you choose a different button in the radio button group a changed_event is generated.

The event is passed the index value of the new choice as an argument. You can use this event to monitor which items get selected in your application. The validity of your radio button group selection may depend upon the state of other controls in the dialog box. In the following example, the changed_event changes the font size of a label in the dialog box.

```
set changed_event(val)
        var object sibling
        sibling = this.sibling@("Label")
        case of val
        case 0  /* Small font */
                sibling.title_font_size@ = 8
        case 1  /* Medium font  */
                sibling.title_font_size@ = 24
        case  2 /* Large font  */
                sibling.title_font_size@ = 48
        endcase
endset
```

See Appendix A, "Sequence of Events", for information about all the events available in the RadioBoxClass.

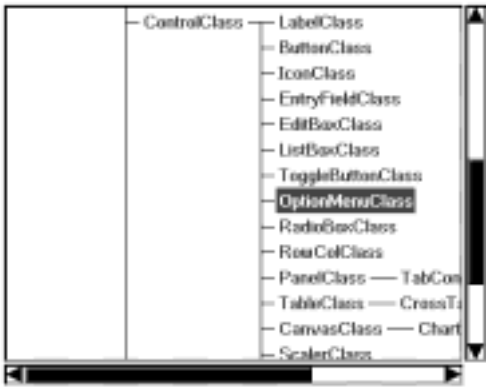
18 Using OptionMenuClass Methods

An option button is a single selection, multiple item control. This chapter covers the following topics:

- OptionMenuClass overview
- Methods and events
- Creating an OptionMenuClass object
- Using an OptionMenuClass object

OptionMenuClass Overview

Use an OptionMenuClass object for choosing a single item from a group of choices. The currently selected item is the only item visible until you choose the option button.



Methods and Events

The following OptionMenuClass methods and events are discussed in this chapter:

- text_strings@
- value_index@
- value@
- changed_event

Creating an OptionMenuClass Object


Designer icon

An OptionMenuClass object is added as an option button control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. To create an option button in a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.

5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Option Button** in the Dialog Box Editor window.
7. Click in the dialog box to place the edit box.
8. Resize the option button and set the option button attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with an option button. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using an OptionMenuClass Object

Use an option button to choose a single item from a group of choices. In addition to setting option button attributes through the Dialog Box Editor window, you can use OptionMenuClass methods and events to set or change:

- information display and access in an option button

- the actions that occur in an option button

OptionMenuClass Information Display and Access

In addition to the methods that set the color and size of the option button, you can use OptionMenuClass methods to control how information is displayed in the option button. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the set method text_strings@ to set the option button items. For example, the following initialize_event for an OptionMenuClass object uses the text_strings@ method to set the option button items.

```
set initialize_event
        this.text_strings@ =
                {"Choice1","Choice2","Choice3"}
endset
```

You can set or determine the item selections for an option button using the set and get methods value@ and value_index@. The set method value@ sets the selected option button selection to the passed string. If the passed string does not match any of the choices, the option button is blank.

The set method value_index@ sets the option button item to the passed index. The index position of option button items in the group is 0-based.

The get methods value@ and value_index@ return the current selection in the option button as, respectively, a string or integer.

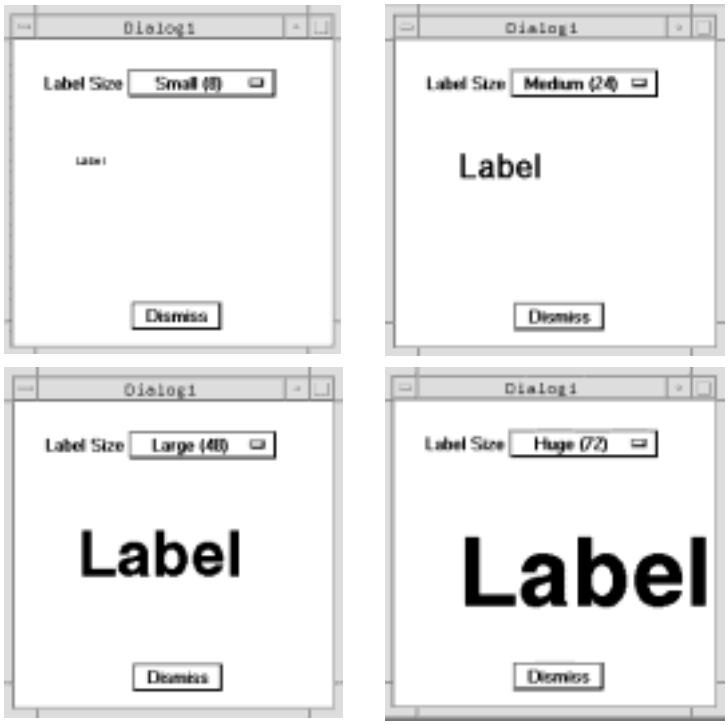
Programming OptionMenuClass Actions

Actions in a OptionMenuClass object generate events. When you choose a different option button item a changed_event is generated.

The event is passed the index value of the new choice as an argument. You can use this event to monitor which items get selected in your

application. The validity of your option button selection may depend upon the state of other controls in the dialog box. In the following example, the changed_event changes the font size of a label in the dialog box.

```
set changed_event(val)
        var object sibling
        sibling = this.sibling@("Label")
        case of val
        case 0  /* Small font */
                sibling.title_font_size@ = 8
        case 1  /* Medium font  */
                sibling.title_font_size@ = 24
        case  2 /* Large font  */
                sibling.title_font_size@ = 48
        case  3 /* Huge font  */
                sibling.title_font_size@ = 72
        endcase
endset
```



See Appendix A, "Sequence of Events", for information about all the events available in the OptionMenuClass.

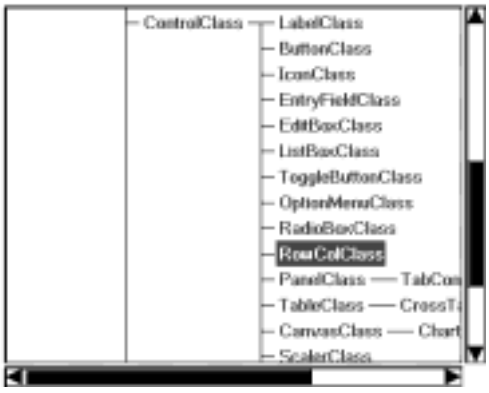
19 Using RowColClass Methods

A RowCol button is a single selection, multiple item control. This chapter covers the following topics:

- RowColClass overview
- Methods and events
- Creating a RowColClass object
- Using a RowColClass object

RowColClass Overview

Use a RowColClass object for choosing a single item from a group of choices. A RowCol button is similar to an option button. Use a RowCol button to display either text or bitmaps in an expandable, muliple column menu. An option button is limited to a single column of text. The currently selected item is the only item visible until you choose the RowCol button.



Methods and Events

The following RowColClass methods and events are discussed in this chapter:

contents_type_is_pixmaps@
contents_type_is_strings@
number_of_columns@
pixmaps@
pixmaps_labels@
style_column_major@
style_row_major@
style_single_column@
style_single_row@
style_square@
style_type@
text_strings@
value@
value_index@
changed_event

Creating a RowColClass Object

An RowColClass object is added as a RowCol button control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box.

Creating a RowColClass object is a two-stage process. To create a RowColClass object, you need to:

- Create an option button in a dialog box using the Designer.

- Change the OptionMenuClass object to a RowColClass object using the Properties dialog box.

To create an option button in a dialog box:

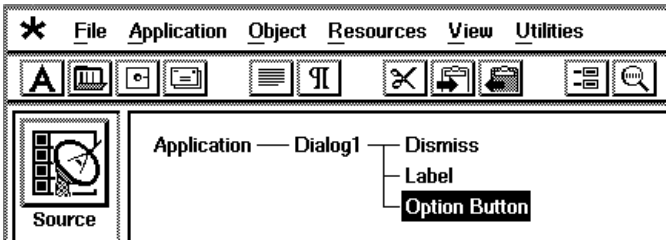

Designer icon

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Option Button** in the Dialog Box Editor window.
7. Click in the dialog box to place the edit box.
8. Resize the option button and set the option button attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

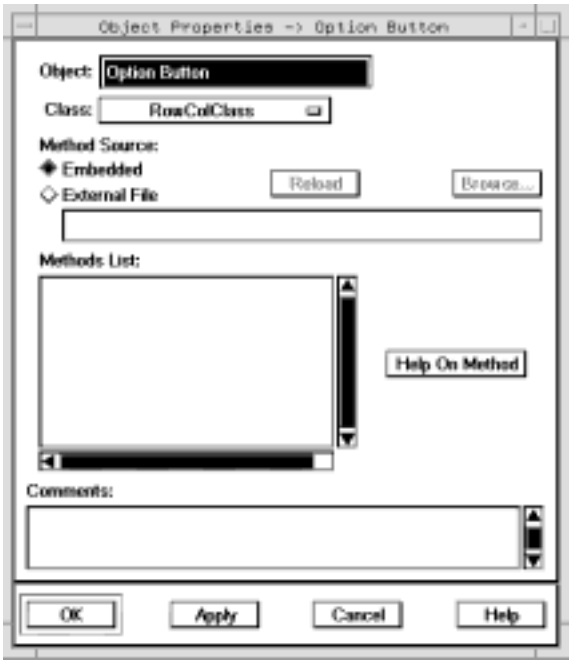
The dialog box is created with an option button. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

After you create the option button in the dialog box, change the class of the option button to RowColClass. To change the class of the option button:

1. Expand the hierarchy in the Browser window to display all children of the dialog box.
2. Select the option button object in the Browser window.



3. Choose **Object → Properties**. The Object Properties dialog box is displayed.



4. Choose **RowColClass** with the Class option to change the inherited class from OptionMenuClass to RowColClass.

5. Click on **OK**.

The option button in the dialog box becomes a RowColClass object. See Chapter 2 in the *Applix Builder User's Guide*, "Using the Browser Tool", for more information about changing object properties.

Using a RowColClass Object

Use a RowCol button to choose a single item from a group of choices. In addition to setting RowCol button attributes through the Dialog Box Editor window, you can use RowColClass methods and events to set or change:

- information display and access in an RowCol button
- the actions that occur in an RowCol button

RowColClass Information Display and Access

In addition to the methods that set the color and size of the RowCol button, you can use RowColClass methods to control how information is displayed in the RowCol button. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

RowCol Colors and Bitmaps

Colors are only available for RowCol buttons that display bitmaps. The background_color_* methods set the background color, and the control_color_* methods set the foreground color. Unlike the other controls, the color settings must be defined for each bitmap in the RowCol button. Each argument for the method is an array of arguments, corresponding to the set bitmap.

For example, to set the foreground color by name for a RowCol button with three bitmaps, you would use the control_color_name@ method as follows:

```
this.control_color_name@ = {"Red","Blue","Red"}
```

RowCol Information

A RowCol button can contain text strings or bitmaps. Use the set methods contents_type_is_pixmaps@ and contents_type_is_strings@ to set the type of information displayed in the RowCol button.

Set contents_type_is_pixmaps@ to FALSE or contents_type_is_strings@ to TRUE to use text strings in the RowCol button. Use the set method text_strings@ to define the text strings used in the RowCol button. The text_strings@ method takes an array of strings as an argument.

Set contents_type_is_pixmaps@ to TRUE or contents_type_is_strings@ to FALSE to use bitmaps in the RowCol button. Use the set method pixmaps@ to define the bitmaps used in the RowCol button. The pixmaps@ method takes an array of bitmap names as an argument.

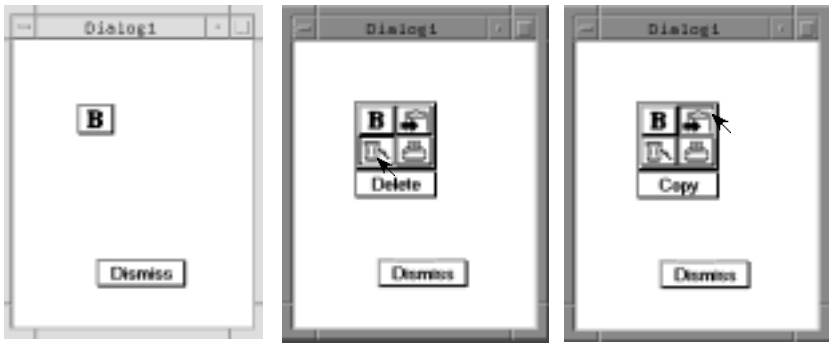
You can use bitmaps in the RowCol button that you create or bitmaps cached in memory by Applix Builder. All of the bitmaps used by Applix Builder and Applixware are defined in a pixmap cache during your Applix Builder or Applixware session. Use the CommonDlgClass get method bitmap_dlg@ to display the Icon Viewer dialog box. Use the dialog box to preview all of the available bitmaps cached in the system.

Use the set method pixmap_labels@ to associate a label with a bitmap. The label is diaplyed when you choose the bitmap in the RowCol button. Use labels to help identify the bitmaps in the RowCol button. For example, the following initialize_event defines four bitmaps for the RowCol button, and four labels that are associated with the bitmaps. The bitamps are defined in the pixmap cache.

```
set initialize_event
        var  pix
        this.contents_type_is_strings@ = FALSE
        pix[0] = "el_bold"
```

```
        pix[1] = "el_copy"
        pix[2] = "el_del"
        pix[3] = "el_print"
        this.pixmaps@ = pix
        this.pixmap_labels@ = {"Bold","Copy","Delete","Print"}
    endset
```

The following figure shows how the bitmaps and labels are displayed in the RowCol button.



In addition to choosing the type of information displayed in a RowCol button, you can also determine the layout of the information with the style methods. The following style types are available:

style_column_major@

Sets the display style of the RowCol button to fill columns first

style_row_major@

Sets the display style of the RowCol button to fill rows first

style_single_column@

Sets the display style of the RowCol button to display the information in a single column, similar to the option button

style_single_row@

Sets the display style of the RowCol button to display the information in a single row

style_square@ Sets the display style of the RowCol button to display rows and columns equally

style_type@ Takes an integer as an argument, sets the RowCol button type. The valid styles are:

0	Square
1	Single column
2	Single row
3	Row major
4	Column major

You can set or determine the item selections for a RowCol button using the set and get methods value@ and value_index@. The set method value@ sets the selected RowCol button selection to the passed string. If the passed string does not match any of the choices, the RowCol button is blank.

The set method value_index@ sets the RowCol button item to the passed index. The index position of RowCol button items in the group is 0-based.

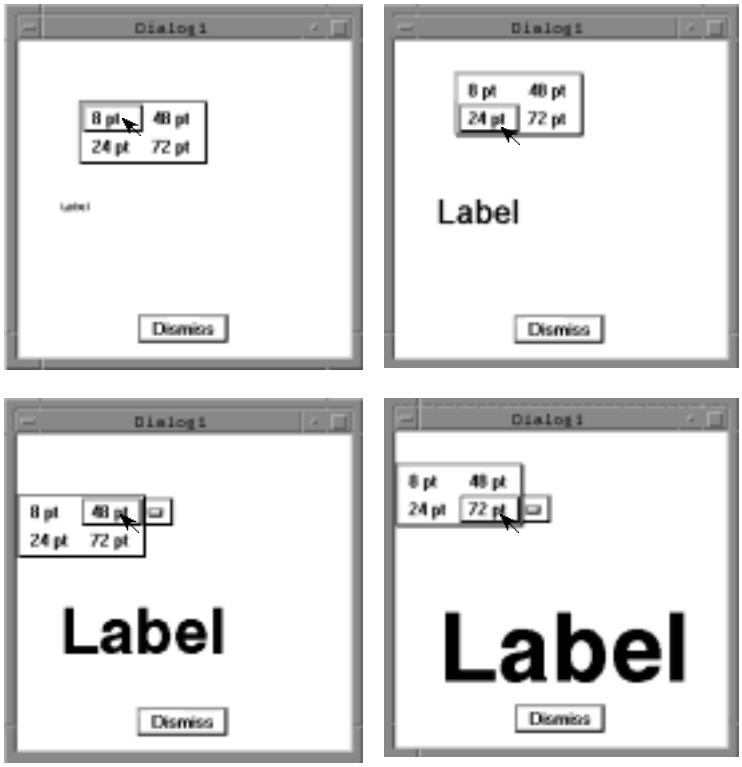
The get methods value@ and value_index@ return the current selection in the RowCol button as, respectively, a string or integer.

Programming RowColClass Actions

Actions in a RowColClass object generate events. When you choose a different RowCol button item a changed_event is generated.

The event is passed the index value of the new choice as an argument. You can use this event to monitor which items get selected in your application. The validity of your RowCol button selection may depend upon the state of other controls in the dialog box. In the following example, the changed_event changes the font size of a label in the dialog box.

```
set changed_event(val)
        var object sibling
        sibling = this.sibling@("Label")
        case of val
        case 0  /* Small font */
                sibling.title_font_size@ = 8
        case 1  /* Medium font  */
                sibling.title_font_size@ = 24
        case  2 /* Large font  */
                sibling.title_font_size@ = 48
        case  3 /* Huge font  */
                sibling.title_font_size@ = 72
        endcase
endset
```



See Appendix A, "Sequence of Events", for information about all the events available in the RowColClass.

20 Using ToggleButtonClass Methods

A toggle button is a state control. This chapter covers the following topics:

- ToggleButtonClass overview
- Methods and events
- Creating a ToggleButtonClass object
- Using a ToggleButtonClass object

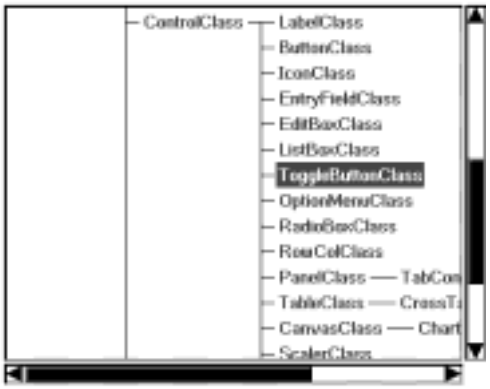
ToggleButtonClass Overview

Use a ToggleButtonClass object to set the state of attributes in your application. A toggle button is either:

- a two state, on-off control

or

- a three state, on-off-mixed control



Methods and Events

The following ToggleButtonClass methods and events are discussed in this chapter:

is_on@	value@
is_three_state@	changed_event
mux_toggle@	

Creating a ToggleButtonClass Object


Designer icon

A ToggleButtonClass object is added as a toggle button control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. To create a toggle button in a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.

6. Choose **Controls → Toggle Button** in the Dialog Box Editor window.
7. Click in the dialog box to place the toggle button.
8. Resize the toggle button and set the toggle button attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a toggle button. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using a ToggleButtonClass Object

Use a toggle button to set the state of attributes in your application. In addition to setting toggle button attributes through the Dialog Box Editor window, you can use ToggleButtonClass methods and events to set or change:

- information display and access in a toggle button
- the actions that occur in a toggle button

ToggleButtonClass Information Display and Access

In addition to the methods that set the color and size of the toggle button, you can use ToggleButtonClass methods to control how information is displayed in the edit box. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the set method is_on@ to set the toggle button state. Set the method to TRUE to turn on the toggle button, set the method to FALSE to turn off the toggle button. Use the get method is_on@ to get the state of the toggle button.

You can also use the set method value@ to set the toggle button state. The method takes an integer as an argument, corresponding to the following states:

0	Toggle button set to off.
1	Toggle button set to on.
2	Toggle button is grayed, available when is_three_state@ is set to TRUE.

Thge toggle button is a two-state button by default. Use the set method is_three_state@ to make the toggle button a three-state button. The three states for a toggle button are on, off, and grayed. Use the grayed state in your application to indicate a mixed or default state of an attribute. Set the method to TRUE to make the toggle button three-state, otherwise set the method to FALSE to make the toggle button a default two-state button.

You can change the toggle button appearance with the set method mux_toggle@. Set the method to TRUE to make the toggle button diamond-shaped. Set the method to FALSE to make the toggle button the default square shape.

Programming ToggleButtonClass Actions

A changed_event is called when a toggle button selection changes. The event is passed the state of the toggle button as an argument. You can use this event to monitor and set the state of your application components. The validity of your toggle button selection may depend upon the state of other controls in the dialog box. The following example sets the label in the dialog box to reflect the toggle button state.

```
set initialize_event
        this.is_three_state@ = TRUE
endset

set changed_event(val)
        var object sibling
        sibling = this.sibling@("Label")
        case of val
        case 0  'Off
                sibling.value@ = "Toggle is Off"
        case 1  'On
                sibling.value@ = "Toggle is On"
        case 2
                sibling.value@ = "Toggle is Grayed"
        endcase
endset
```

See Appendix A, "Sequence of Events", for information about all the events available in the ToggleButtonClass.

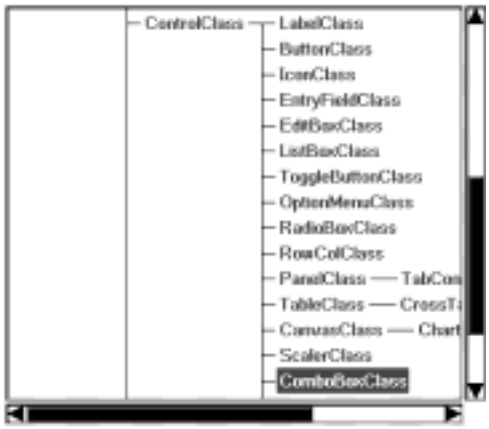
21 Using ComboBoxClass Methods

A combo box is a single selection, multiple item control. This chapter covers the following topics:

- ComboBoxClass overview
- Methods and events
- Creating a ComboBoxClass object
- Using a ComboBoxClass object

ComboBoxClass Overview

Use a ComboBoxClass object for choosing a single item from a group of choices. The currently selected item is the only item visible until you choose the combo box, then all choices are displayed. All choices remain visible until you choose an item or click elsewhere on the screen.



Methods and Events

The following ComboBoxClass methods and events are discussed in this chapter:

text_strings@	value_index@
value@	changed_event

Creating a ComboBoxClass Object

A ComboBoxClass object is added as a combo box control in a dialog box. Use the Dialog Box Editor window to add and position controls in a dialog box. To create a combo box in a dialog box:


Designer icon

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Combo Box** in the Dialog Box Editor window.

7. Click in the dialog box to place the combo box.
8. Resize the combo box and set the combo box attributes using the dialog box editing options.
9. Choose **File → Save** when you are through editing the dialog box to save all edits.
10. Choose **File → Exit**.
11. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a combo box. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4 in the *Applix Builder User's Guide*, "Using the Designer Tool", for more information on creating and editing dialog boxes.

Using a ComboBoxClass Object

Use a combo box to choose a single item from a group of choices. You can use a combo box wherever you would use an option button in an application. The advantage of using a combo box is that it displays all items when you click on it, until you make a selection. With an option button you must keep the mouse button pressed to display all items. In addition to setting combo box attributes through the Dialog Box Editor window, you can use ComboBoxClass methods and events to set or change:

- information display and access in a combo box
- the actions that occur in a combo box

ComboBoxClass Information Display and Access

In addition to the methods that set the color and size of the combo box, you can use ComboBoxClass methods to control how information is displayed in the combo box. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the set method text_strings@ to set the combo box items. The method must receive an array of strings as an argument. For example, the following initialize_event for a ComboBoxClass object uses the text_strings@ method to set the combo box items.

```
set initialize_event
        this.text_strings@ =
                {"Choice1","Choice2","Choice3"}
endset
```

You can set or determine the item selections for a combo box using the set and get methods value@ and value_index@. The set method value@ sets the selected combo box selection to the passed string. If the passed string does not match any of the choices, the combo box is blank.

The set method value_index@ sets the combo box item to the passed index. The index position of combo box items in the group is 0-based.

The get methods value@ and value_index@ return the current selection in the combo box as, respectively, a string or integer.

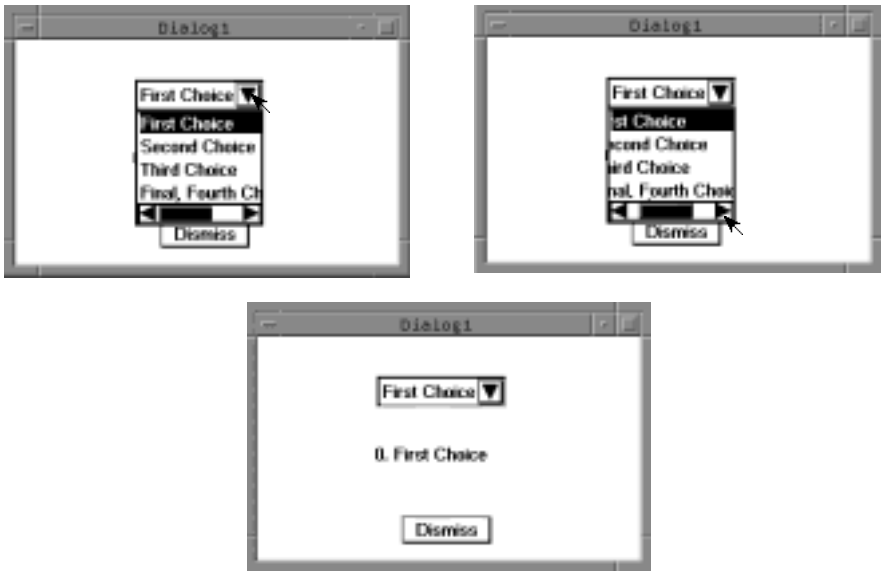
Programming ComboBoxClass Actions

Actions in a ComboBoxClass object generate events. When you choose a different combo box item a changed_event is generated.

The event is passed the index value of the new choice as an argument. You can use this event to monitor which items get selected in your

application. The validity of your combo box selection may depend upon the state of other controls in the dialog box. In the following example, the changed_event sets the value of a label in the dialog box to the chosen item, prefaced with the item index.

```
set changed_event(ind)
        var object sibling
        sibling = this.sibling@("Label")
        sibling.value@ = ind ++". "++this.value@
endset
```



Notice that if the length of an item is greater than the width of the combo box, a horizontal scroll bar appears. You can use the scroll bar to see an entire item.

See Appendix A, "Sequence of Events", for information about all the events available in the ComboBoxClass.

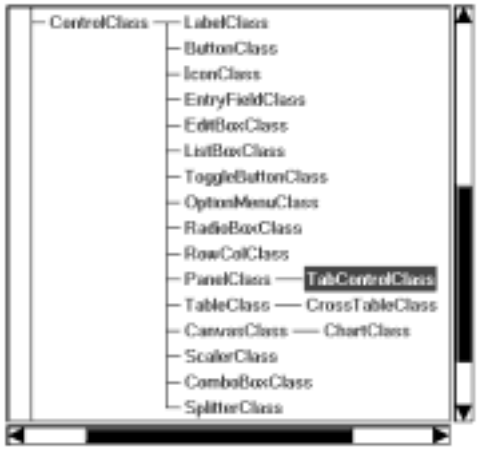
22 Using TabControlClass Methods

A tab control allows you to create a multiple-layered dialog. This chapter covers the following topics:

- TabControlClass overview
- Methods and events
- Creating a TabControlClass object
- Using a TabControlClass object

TabControlClass Overview

Use a TabControlClass object to create a multiple-layered dialog box. The tab control handles the hiding and displaying of controls in the tab dialog box, depending upon which layer you choose. The TabControl-Class is based upon the PanelClass. It inherits all the PanelClass methods, use these methods to programmatically set size and color attributes of the tab control.



Methods and Events

The following TabControlClass methods and events are discussed in this chapter:

active_layer@	load_dbox@
containees@	top_inset@
insert_controls@	tab_event
layernames@	

Creating a TabControlClass Object

Creating a TabControlClass object is a two-stage process. To create a TabControlClass object, you need to:

- Create a panel in a dialog box using the Designer.
- Change the PanelClass object to a TabControlClass object using the Panel Attributes dialog box.

To create a panel in a dialog box:

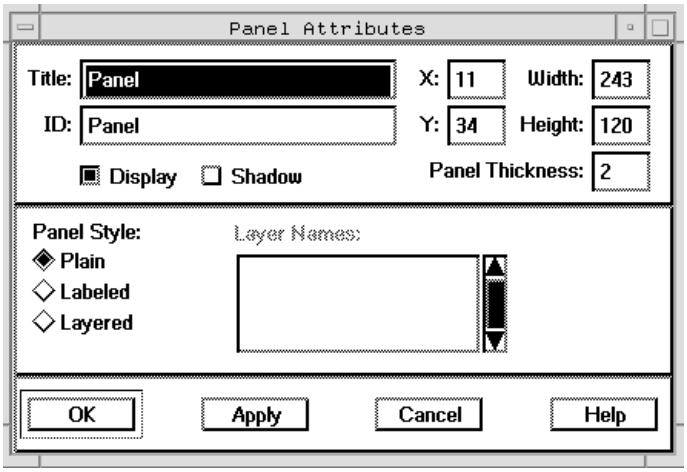

Designer icon

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.

4. Choose the type of dialog box with the Type option.
5. Click on **Edit**. The dialog box appears in a Dialog Box Editor window.
6. Choose **Controls → Panel** in the Dialog Box Editor window.
7. Click in the dialog box to place the panel.
8. Resize the panel and set the panel attributes using the dialog box editing options.

After you create the panel in the dialog box, use the Panel Attributes dialog box to change the panel to a tab control. To create a tab control:

1. Double-click on the panel. The Panel Attributes dialog box appears.

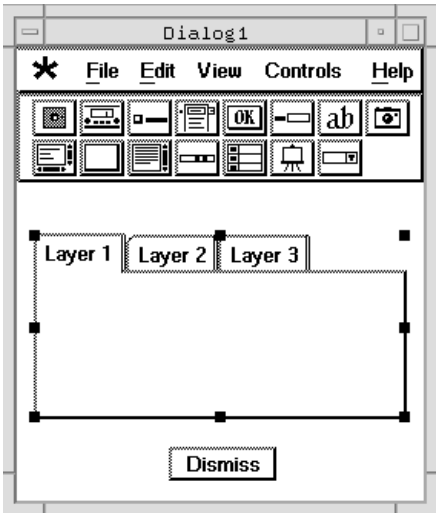


2. Choose **Layered** from the Panel Style radio group.

   The Layer Names edit box is ungrayed, and default layer names are inserted in the edit box.

3. Click in the Layer Names edit box and type the layer names, one layer name per line.

4. Click **OK** in the Panel Attributes dialog box. The panel is changed to a tab control in the dialog box.



5. Put dialog box controls on the tab layers. You can tab between layers, only the controls on the selected layer are displayed.

6. Choose **File → Save** when you are through editing the dialog box to save all edits.

7. Choose **File → Exit**.

8. Click on **Dismiss** in the Designer dialog box.

The dialog box is created with a tab control. The dialog box appears in the Browser window, if you expand the application hierarchy. See Chapter 4, "Using the Designer Tool", for more information about creating and editing dialog boxes.

Using a TabControlClass Object

Use a TabControlClass object to create a multiple-layered dialog box. Use the tab control to manage the display of controls, instead of programmatically hiding and displaying controls in the dialog box. Controls on hidden layers still exist in the dialog box, you can use their settings and attributes in your application execution.

In addition to setting tab control attributes through the Dialog Box Editor window, you can use TabControlClass methods and events to set or change:

- the tab control display
- the actions that occur in a tab control

TabControlClass Display

In addition to the methods that set the color and size of the tab control, you can use TabControlClass methods to control how information is displayed in the tab control. See Chapter 7, "Using Dialog Box Controls", for information about setting control colors.

Use the get method top_inset@ to get, in pixels, the height of the tab area. Use this value to determine the placement of a control on a layer. The tab control's y-position, plus the tab height, is the highest area on the panel for a control y-position.

Use the set method layernames@ to set the layers in the tab control. The method takes an array of strings as an argument. Use the corresponding get method to get the names of all layers in the tab control.

Use the get method containees@ to return references to controls in the layer. The method takes the name of a layer as an argument, if a name is not supplied then all controls in the tab control are returned. The control references are returned as an array of objects. Use this method to determine if a control is appearing in the correct layer, or that unwanted controls do not exist on the layer.

Use the set method active_layer@ to set the active tab control layer. The method takes the name of the layer as an argument. If the passed layer name does not exist, no change is made to the display of the tab control layers.

Use the get method active_layer@ to return the active tab control layer. The method returns the name of the active layer as a string.

You cannot use either method in an initialize_event, you can only use them when the control is displayed, such as in an update_event. For example, the following update_event for a TabControlClass object uses the active_layer@ method to set the tab control layer to Layer 1.

```
set update_event
        IF this.active_layer@ <> "Layer 1"
                this.active_layer@("Layer 1")
endset
```

Use the set method load_dbox@ to load the contents of a dialog box into a layer. The method takes two arguments:

- the layer name
- the name of the dialog box, including the .d file extension

You must supply the full path name for the file if the dialog box is not in a directory defined in your ELF search list. The dialog box controls only exist dynamically on the layer, the controls are not saved on the layer when you exit the application. The dialog box you load should have the same dimensions as the tab control, and the y-position of controls in the dialog box should not be less than the y-position of the

tab control plus the value returned by top_inset@, so that the controls are displayed properly in the tab control layer.

Use this method when you are working with a tab control that has many layers. Instead of creating controls for each layer, and waiting for them to be initialized when the layer becomes active, you can load the controls in one step. This will improve the performance of your application. Use the Applixware Dialog Box Editor to create dialog boxes that you can load into a tab control layer.

Use the set method insert_control@ to add individual controls to a dialog. The method takes two arguments:

- the layer name
- the object to insert in the layer

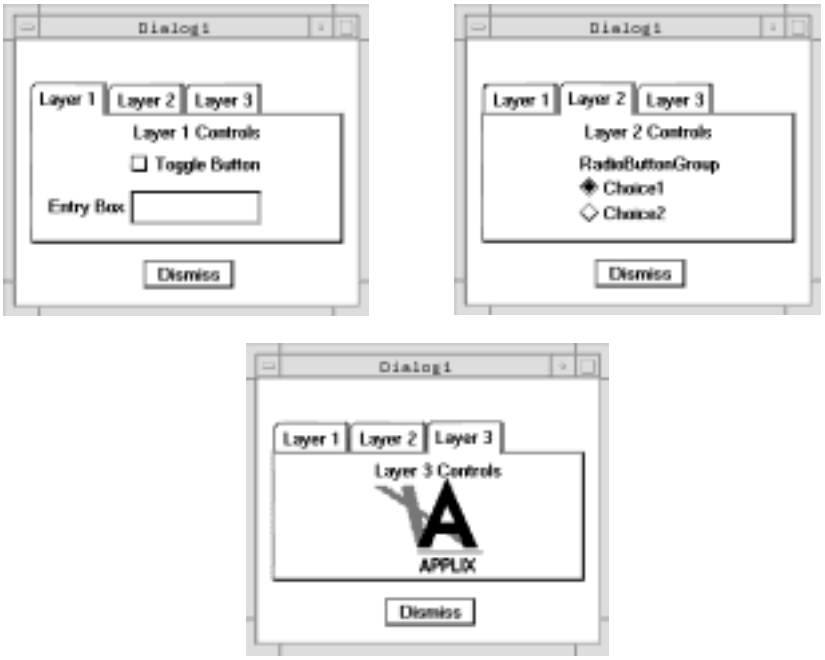
Use this method if you want to load only a few controls in a layer, and not an entire dialog box. You must still set the coordinates of the control so that it appears within the layer, as well as any other control attributes.

Programming TabControlClass Actions

Actions in a TabControlClass object generate events. When you choose a different tab control layer a tab_event is generated.

The event is passed the name of the active layer as an argument. You can use this event to manage the controls in the layer, such as using the load_dbox@ method or setting the state of controls on the layer. In the following example the tab_event loads a dialog box into the active layer,

```
set tab_event(name)
        case of name
        case "Layer 1"
                this.load_dbox@(name, "layer1.d")
        case "Layer 2"
                this.load_dbox@(name, "layer2.d")
        case "Layer 3"
                this.load_dbox@(name, "layer3.d")
        endcase
endset
```



See Appendix A, "Sequence of Events", for information about all the events available in the TabControlClass.

Examples

TabControlClass examples are available in the tab directory. Choose Utilities → Sample Applications to access the tab directory or other sample application directories with the Directory Displayer dialog box. The examples provide an introductory usage of TabControlClass methods and events covered in this chapter.

23 Using SpreadsheetsClass, WordsClass, and GraphicsClass Methods

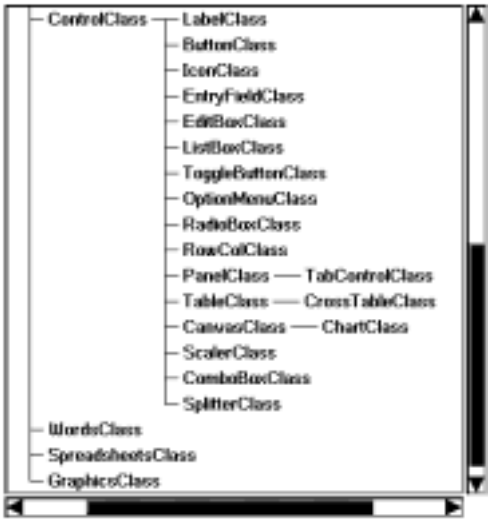
The SpreadsheetsClass, WordsClass, and GraphicsClass allow you to use the corresponding Applixware applications in your Applix Builder application. This chapter covers the following topics:

- Overview
- Using a Applixware objects
- Application Events

Overview

Your Applix Builder applications may require the information created in or the functionality provided by Applix Spreadsheets, Words, or Graphics. Use the SpreadsheetsClass, WordsClass, and GraphicsClass methods to provide total integration with Builder and Applixware applications.

Use the SpreadsheetsClass to perform complex computations and display the information in a chart. Use the WordsClass to create, format, and display text documents. Use the GraphicsClass to draw, edit, and display graphical objects.



.

Using Applixware Objects

Use a SpreadsheetsClass object to organize and analyze your numerical data. Use a WordsClass object to create text-based compound documents. Use a GraphicsClass object to create graphics-based documents. The classes contain many methods to insert, delete, and manipulate information in a document. The classes also contain methods to control the communication between the Applix Builder application and the document.

The document communication can be implemented in two ways:

- an Applix Builder application launching a document
- a document launching an Applix Builder application

SpreadsheetsClass objects also have a set of events that you can program. The events parallel the callback macros you can set in a spreadsheet.

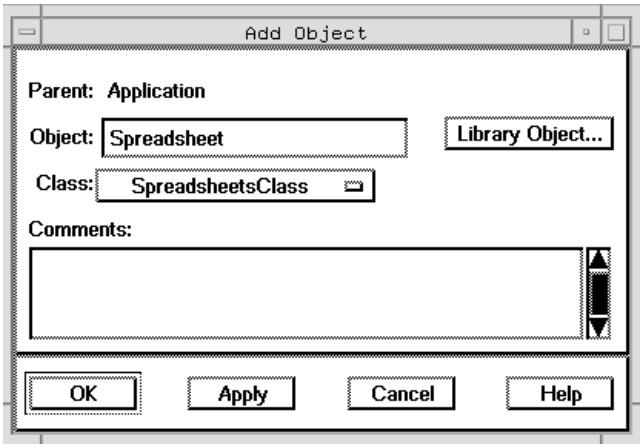
An Application Launching a Document

An Applix Builder application can launch a spreadsheet, words document, or graphics document using methods in the respective class. You can launch a document by adding an object to an application. For a spreadsheet you can also open a new spreadsheet window from an existing SpreadsheetsClass object.

Adding an Object to an Application

You can add a SpreadsheetsClass, WordsClass, or GraphicsClass object as a child of the top-level ApplicationClass object. The following steps use a SpreadsheetsClass object as an example. To add an object:

1. Choose the ApplicationClass object in the main Browser window.
2. Choose **Object → Add**. The Add Object dialog box appears.
3. Type the name of the object in the Object entry area.
4. Choose **SpreadsheetsClass** from the Class option.



The SpreadsheetsClass object is added to the application. To make the spreadsheet visible during application execution, you must:

1. Set the is_windowless@ method to FALSE.
2. Perform an action in the document.

A document is in windowless mode by default. You can use a document in windowless mode to perform actions in the background that are not visible to the application user. If you want the document window to be visible, you must set is_windowless@ to FALSE before you perform any other actions in the document. To make the document appear on screen, you must perfom some action with the object, such as loading a file or moving the cursor within the document. For example, the following initialize_event in the SpreadsheetsClass object would make the spreadsheet visible and move the cursor to cell A1 on the current sheet.

```
set initialize_event
        this.is_windowless@ = FALSE
        this.go_begin@
endset
```

You can assign a menu bar to a document with the set method menu_bar_id@. The method takes two arguments: id, the number identifying the menu bar; and mbInfo, which is either the name of the file containing the menu bar definition, or an array with the actual menu bar definition. The method changes the menu bar for an open document, or initializes the menu bar for a spreadsheet that is not displayed.

Opening a New Spreadsheet Window

You can open a new spreadsheet window from an existing SpreadsheetsClass object with the get method new_win_env@. The method creates a new SpreadsheetsClass object, displays the spreadsheet, and returns a reference to the object. If you use this method you must write the code to add the object to the application or destroy the object when you are finished using it.

The new_win_env@ method takes three arguments:

- a menu bar ID number
- a mouse pointer type
- a menu bar file name

You can use this method to display multiple spreadsheet windows, each with a different menu bar and mouse pointer style. In the following example, the SpreadsheetsClass object's initialize_event displays the spreadsheet window, then it launches a new spreadsheet window using the default menu bar and inheriting the same mouse pointer. The terminate_event destroys the secondary SpreadsheetsClass object created in the initialize_event.

```
objvar object ss2
set initialize_event
        this.is_windowless@ = FALSE
        this.go_begin@
        ss2 = this.new_win_env@(NULL,"inherit","ax_ss4")
endset

set terminate_event
        OBJECT_DESTROY@(ss2)
endset
```

A Document Launching an Application

Applix Spreadsheets, Words, or Graphics can launch an Applix Builder application using the BUILDER_APPLICATION@ macro. To provide integrated communication between the document and the application you need to:

- pass the task ID of the document to the application

- attach an object to the task ID

You can pass the task ID of the document as an argument to the BUILDER_APPLICATION@ macro. For example, the following macro is designed to be called as a menu bar choice from the document. The macro calls the BUILDER_APPLICATION@ macro with file name and the task ID of the document, retrieved with the MACRO_PARENT_TASK@ macro. If you do not call the macro as a menu bar choice you will have to use other ELF macros, such as TASK_LIST@, to get the task ID of the document.

```
macro start_builder_application
  builder_application@("/user/applix/test.ab", macro_parent_task@())
endmacro
```

You need to program your Applix Builder application to use the passed task ID to attach a SpreadsheetsClass, WordsClass, or GraphicsClass object to the calling document. Use the get method attach_to_task@ to attach an object to a document. The method takes the document task ID as an argument, and returns a Boolean value. The method returns TRUE if the it can attach to the document task. The method returns FALSE if the object cannot attach to the task ID, such as passing the task ID to the method for a document that is incompatible with object.

The following example shows how you can program the Spread-sheetsClass object initialize_event to attach to the task ID of a calling spreadsheet.

```
set initialize_event
        var taskId
        taskId = this.application@.arg_var@(0)
        if this.attach_to_task@(taskId) {
                return
        }
        else {
                beep@()
                info_message@("Couldn't Attach to task
"++taskId)
                this.application@.quit@
        }
endset
```

After you attach the object to an open document, the document is capable of receiving method calls and, in the case of a spreadsheet, generating events through the object.

Application Events

The GraphicsClass, SpreadsheetsClass, and WordsClass contain events to help you monitor document actions. The SpreadsheetsClass contains additional events for spreadsheet actions.

GraphicsClass, SpreadsheetsClass, and WordsClass Events

The events common to the GraphicsClass, SpreadsheetsClass, and WordsClass are:

document_exit_event

Called when closing a document.

document_modified_event

Called when saving a document.

document_open_event

Called when opening a document.

task_exit_event

Called when exiting an application window.

SpreadsheetsClass Events

The SpreadsheetsClass contains events, in addition to the standard events, that you can program as callbacks for actions in the spreadsheet. The events are:

cr_event	Called on a carriage return in the spreadsheet. Program this event as you would program the SS_SET_CR_CALLBACK@ macro.
dbl_click_event	Called on a double-click in a cell. Program this event as you would program the SS_SET_DBL_CLICK_CALLBACK@ macro.
load_cell_event	Called when a new value is loaded in a cell. Program this event as you would program the SS_SET_LOAD_CELL_CALLBACK@ macro.
selection_event	Called on a selection in the spreadsheet. Program this event as you would program the SS_SET_CALLBACK@ macro.

24 Building Sample Applications

Using Applix Builder, you can construct a simple application in a few steps with minimal programming. This chapter gives you step-by-step instructions on creating sample applications. This chapter covers the following topics:

- Building a data set application
- Building a sample Real Time application

Overview, Goals

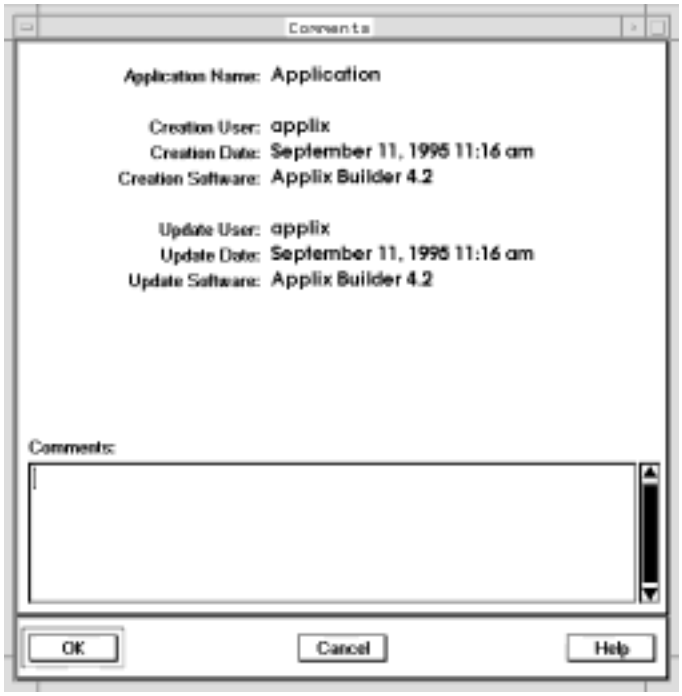
You can create quick, customizable applications using Applix Builder. This chapter focuses on the steps needed to create an introductory, database, or Real Time application. Although this chapter focuses on data-centric applications, you can create applications that do not access database or Real Time information.

When you complete this chapter you will understand the basic steps for creating an application. The sample applications created in this chapter are complete, working products.

Commenting the Application

An application may go through multiple revisions before becoming a deployable application. Use the Comments dialog box to save information and comments about the application. Type all your information, such as creation, edit reasons, or purpose, in the Comments area.

Choose **View → Comments** to add comments to an application.



Previous comments appear in the Comments area. Type your comments in the area. Added comments are saved when you save the application.

Building a Data Set Application

This section shows how to build a data set application. A data set is an SQL query on one or more database tables created with a Source data window. This section covers:

- Establishing a data set
- Creating a dialog box
- Running the application
- Saving and exiting

Requirements

The examples in this section are based on a set of sample database tables. To create data sets and install the sample tables you need:

- An Applix database gateway or vendor gateway
- axnet files for the database server, if the machine is different from the client machine on which you installed Applix Builder

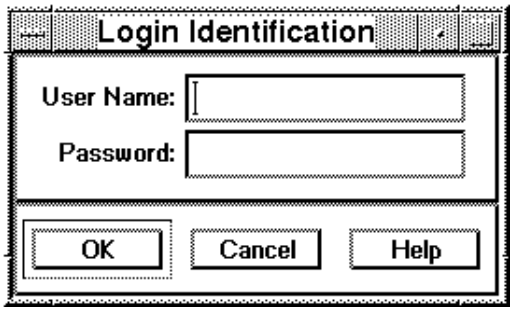
See Chapter 6, "Applixware Communications", in the *Applixware Linux System Administrator's Guide* for information about axnet, database gateway configurations, and installing axnet and database gateways.

Installing Sample Database Tables

To install the sample database tables:

1. Click on the Source icon in the Browser window. The Data Source dialog box appears.
2. Type the data source name in the Data Source entry area.
3. Click on **Create**. The data source name appears in the Data Source list area.
4. Choose **Data Set** from the Source Type radio group.
5. Click on **Edit**. The Source data window for the data set appears.
6. Choose **Query → Choose Server** in the data window.
7. Click on **Add**.
8. Choose a database vendor-type from the Vendor options.
9. Type the database name in the Database entry area.
10. Type the name of the computer on which the database server is running in the Host entry area.
11. Depending on the vendor-type, you may have to enter information in the other entry areas. Information is not required for areas with grayed titles.
12. Click on **OK** to add the database and connect to the database.

    To connect to some vendor databases you must enter a user name and password. Choose Tools → Login Identification to enter your database user name and password.



When a link is established to the database the Ready prompt appears in the status line at the bottom left of the Source data window. If the database information is incorrect, or if you do not have permissions to the database, an error message appears.

13. Choose **Help → Install Demo Tables** in the data window.

    Status messages appear in the status line of the Source data window while tables are installing. The message "Demo Tables Installed" appears in the status line when the installation is complete.

Creating a Data Set

You create a data set with the Source tool. To create a data set:

1. Click on the Source icon in the Browser window. The Data Source dialog box appears.
2. Type the name of the data set in the Data Source entry area. The default is Data1.
3. Click on **Create**. The data set name appears in the Data Source list area.
4. Choose **Data Set** from the Source Type radio group.

5. Click on **Edit**. The Source data window for the data set appears.
6. Choose **Query → Choose Server** in the data window.

   The Choose Server dialog box appears. If the List of Database Connections list area does not contain the database server and gateways you want to connect to, see Chapter 3 in the *Applix Builder User's Guide*, "Using the Source Tool", for information about adding a database.
7. Double-click on a database name in the List of Database Connections list area.
8. Choose **Query → Choose Tables** in the data window. The Choose Tables dialog box appears.
9. Double-click on the axsupplier table in the Available Tables list area. The table appears in the Tables to Query list area.
10. Click **OK** in the Choose Tables dialog box.
11. Choose **File → Save** in the data window.
12. Choose **File → Exit** in the data window.

You can type comments with information for the data source, purpose of creation, and so on in the Comments area of the Data Source dialog box. When you have completed your comments, click Dismiss to dismiss the dialog box.

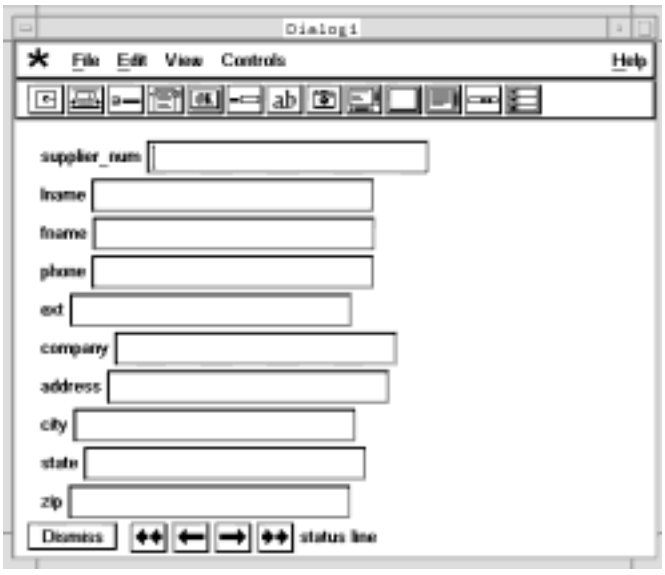
Creating a Dialog Box

You create a dialog box with the Designer tool. Every application requires at least one dialog box. To create a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.

2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**. The dialog box name appears in the Dialog Boxes list area.
4. Choose the data set created in the previous section with the Data Set option.
5. Choose **Main** in the Type option.
6. Click on **Edit**.

   The designer window appears, with an entry area for each column of the data set and push buttons for accessing the database information. The controls are automatically created when you choose a data set with a dialog box before you edit the dialog box for the first time.



   You can move, modify, or delete the controls in the designer window. You can add controls in the window, or even change the

size and title of the window itself. When you are through editing in the designer window:

7. Choose **File → Save**.

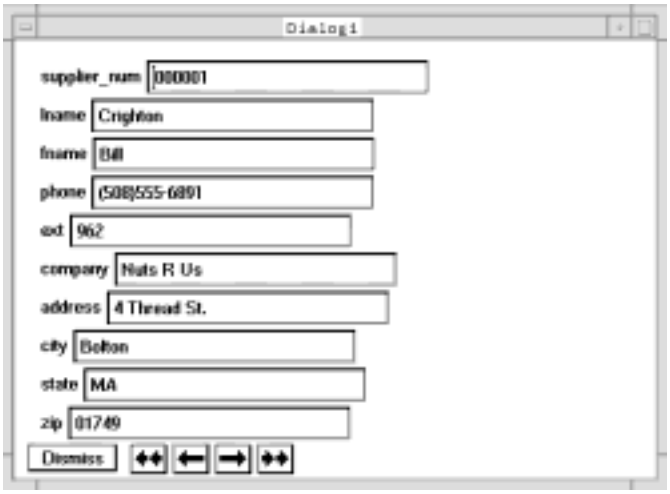
8. Choose **File → Exit**.

You can type comments with information for the dialog box, purpose of creation, and so on in the Comments area of the Designer dialog box. When you have completed your comments, click Dismiss to dismiss the dialog box.

Running an Application

You can run an application from the Builder main menu. You can run an application while you are working on it, allowing you to make changes and fine tune your application.

To run an application, click on the Run button on the Builder main menu. The main application dialog box appears.



Saving and Exiting

Choose File → Save to save the current application. The first time you save an application, File → Save As appears so you can specify a name, location, format, and permissions.

To close the application, choose File → Exit.

If you made changes since the last save, you get a chance to: save the edits; discard them but still close the application; or cancel the exit and return to the application without making any of the changes.

If you are just trying to open another application, this message appears when the application you are closing has changed since the last save. You must indicate what should be done with those edits (Save, Discard, or Cancel) before the Open option displays.

See "Application File Options" in Chapter 2 of the *Applix Builder User's Guide*, "Using the Browser Tool" for information on opening, saving, and exiting an application.

Building a Real Time Application

This section shows how to build an application with a Real Time data source. You can also create applications without a data source, or with a data set.

- Creating a Real Time data feed
- Creating a dialog box
- Connecting a dialog box

- Running the application
- Saving and exiting

Requirements

The examples in this section are based on the rtdemo sample Real Time gateway. To use other gateways you need the appropriate Real Time gateway installed and licensed.

Creating a Real Time Data Source

Create a Real Time data source with the Source tool. The following procedures apply to all Real Time Gateways.

To create a data feed with the Data Source dialog box:

1. Click on the Source icon in the Browser window. The Data Source dialog box appears.
2. Type the data feed name in the Data Source entry area.
3. Click on **Create**. The data source name appears in the Data Source list area.
4. Choose **Real Time** from the Source Type radio group.
5. Click on **Edit**. The Edit Real Time Gateway dialog box appears.
6. Type **rtdemo** in the Gateway entry area. The gateway is the name of the Real Time engine.
7. Type **demo** in the Records entry area. Use this name to reference the record in the Browser and Connector.

8. Click on **Add**. The record name appears in the Records list area.

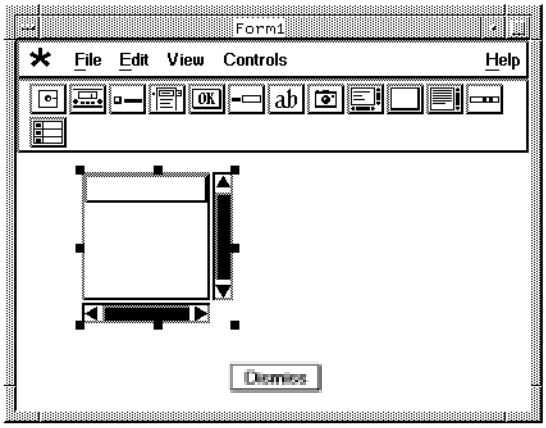
9. Click on **Edit**. The Edit Real Time Record dialog box appears.
10. Type **DEM=** in the Record ID entry area.
11. Type **IDN_SELECTFEED** in the Service Name entry area.
12. Type **DSPLY_NAME** in the Field Name entry area.
13. Click on **Add**.
14. Type **BID** in the Field Name entry area.
15. Click on **Add**.
16. Type **ASK** in the Field Name entry area.
17. Click on **Add**.
18. Click on **Dismiss** in the Edit Real Time Record dialog box.
19. Click on **Dismiss** in the Edit Real Time Gateway dialog box.

You can type comments with information for the data source, purpose of creation, and so on in the Comments area of the Data Source dialog box. When you have completed your comments, click on Dismiss to dismiss the dialog box.

Creating a Dialog Box

Create a dialog box with the Designer tool. Every application requires at least one dialog box. For a Real Time data source, use an entry box or table to present the information. To create a dialog box:

1. Click on the Designer icon in the Browser window. The Designer dialog box appears.
2. Type the name of the dialog box in the Dialog Boxes entry area.
3. Click on **Create**.
4. Choose **Main** in the Type option.
5. Click on **Edit**. The designer window appears.
6. Choose **Controls → Table** and click in the designer window.



A table appears in the designer window. Click and drag on the handles to resize the table in the dialog box.

You can move, modify, or delete the controls in the designer window. You can add controls in the window, or even change the size and title of the window itself.

When you are through editing in the designer window:

7. Choose **File → Save**.
8. Choose **File → Exit**.

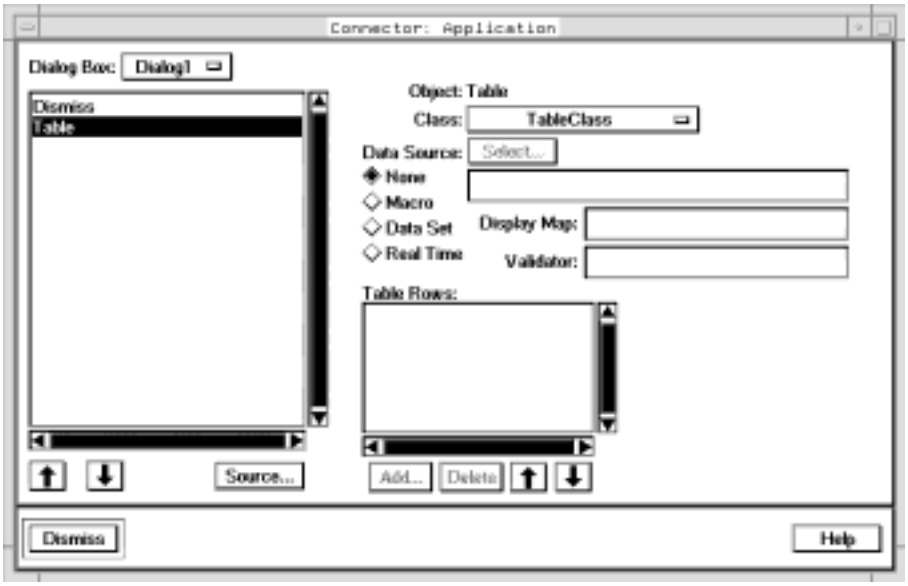
You can type comments with information for the dialog box, purpose of creation, and so on in the Comments area of the Designer dialog box. When you have completed your comments, click Dismiss to dismiss the dialog box.

Connecting a Dialog Box

After you create a dialog box and a data source, use the Connector to connect the dialog box components to the data source. The following

steps are for connecting a table. See "Creating a Real Time Table" in Chapter 5 of the *Applix Builder User's Guide*, "Using the Connector Tool", for information about connecting tables and Real Time data sources. To connect a dialog box and a data source:

1. Click on the Connector icon in the Browser window. The Connector dialog box appears.
2. Choose the dialog box created in the previous section with the Dialog Box option.
3. Click on **Table** in the Dialog Box list area.



The ID of the selected object appears adjacent to the Object label and the inherited class of the objects appears in the Class option.

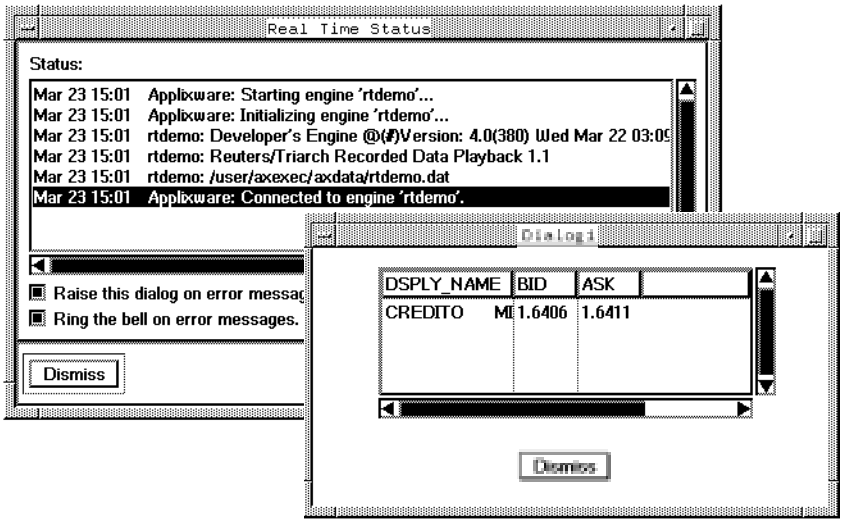
4. Choose **Real Time** from the Data Source radio group.

5. Click on **Select**. The Choose Data Feed Row and Field dialog box appears.
6. Click on **demo** in the Rows list area.
7. CTRL-Click on **DSPLY_NAME** in the Fields list area.
8. CTRL-Click on **BID** in the Fields list area.
9. CTRL-Click on **ASK** in the Fields list area.
10. Click on **OK**.
11. Click on **Dismiss** in the Connector dialog box when you finish connecting entry box components to Real Time data sources.

Running an Application

You can run an application from the Builder main menu. You can run an application while you are working on it, allowing you to make changes and fine tune your application.

To run an application, click on the Run button on the Builder main menu. The main application and Real Time Status dialog boxes appear.



Saving and Exiting

Choose File → Save to save the current application. The first time you save an application, File → Save As appears so you can specify a name, location, format, and permissions.

To close the application, choose File → Exit.

If you made changes since the last save, you get a chance to: save the edits; discard them but still close the application; or cancel the exit and return to the application without making any of the changes.

If you are just trying to open another application, this message appears when the application you are closing has changed since the last save. You must indicate what should be done with those edits (Save, Discard, or Cancel) before the Open option displays.

A Sequence of Events

This appendix lists the events called in the DialogBox-Class and all the dialog box control classes. A sequence of events is listed for situations where multiple event calls may occur for a given control action.

Sequence of Events

Application objects have different types of events you can program to determine application execution. The "Object Events" section in Chapter 2, "Object-Oriented Concepts", discusses the event sequence for application initialization and termination. Dialog box controls have events that control the interaction with the object. For example, a ButtonClass object has a clicked_event that you program to define the actions that occur when the button is clicked. This section lists the events available in each object, when each event is called, and the sequence of events for multiple event calls.

This section lists the control classes in alphabetical order after the listing of events for DialogBoxClass and the decoration controls. The events for each dialog box control are called in the order that they appear in the Browser.

DialogBoxClass

error_event	Called for error with dialog box.
destroy_event	Called before the dialog box is destroyed by the window manager.
initialize_event	Called before the dialog box is displayed. The dialog box initialize_event is called before the initialize_events for its children controls.
poke_event	Called when the dialog box receives a poke code defined with the accept_pokes@ method.
resize_event	Called when the dialog box is resized.

terminate_event	Called after the dialog box is dismissed.
time_out_event	Called when the time limit set in the object with the timer@ method expires.

IconClass, LabelClass, PanelClass

IconClass, LabelClass, and PanelClass objects are decorations, the events associated with these classes is limited to:

error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

ButtonClass

clicked_event	Called when the button is clicked.
error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.

resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

CanvasClass

button2_double_click_event

Called when the middle mouse button is double-clicked. The event is passed the point of the double-click as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values. The sequence of event calls for a middle mouse button double click is:

1. button2_press_event
2. button2_release_event
3. button2_double_click_event
4. button2_release_event

A button2_motion_event is called if the mouse is moved before either button2_release_event. A button2_motion_event is called for each mouse movement.

button2_motion_event

Called when the mouse is moved while the middle mouse button is pressed. The event is passed the end point of the motion as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

The button2_motion_event can occur in or out of the canvas area. If the event occurs outside of the canvas area the point may contain negative values, or values greater than the canvas area dimensions.

button2_press_event

Called when the middle mouse button is pressed and the mouse pointer is in the canvas area. The event is passed the point of the button press as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

button2_release_event

Called when the middle mouse button is released. The event is passed the point of the button release as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

The button2_release_event can occur in or out of the canvas area. If the event occurs outside of the canvas area the point may contain negative values, or values greater than the canvas area dimensions.

button3_double_click_event

Called when the right mouse button is double-clicked. The event is passed the point of the double-click as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean

values. The sequence of event calls for a right mouse button double click is:

1. button3_press_event
2. button3_release_event
3. button3_double_click_event
4. button3_release_event

A button3_motion_event is called if the mouse is moved before either button3_release_event. A button3_motion_event is called for each mouse movement.

button3_menu_state_event

Called before a popup menu is displayed. A popup menu is programmed with the button3_menu_info@ method. If menu info is set, the button3_double_click_event, button3_motion_event, button3_press_event, and button3_release_event are disabled.

button3_motion_event

Called when the mouse is moved while the right mouse button is pressed. The event is passed the end point of the motion as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

The button3_motion_event can occur in or out of the canvas area. If the event occurs outside of the canvas area the point may contain negative values, or values greater than the canvas area dimensions.

button3_press_event

Called when the right mouse button is pressed and the mouse pointer is in the canvas area. The event is passed the point of the button press as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

button3_release_event

Called when the right mouse button is released. The event is passed the point of the button release as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

The button3_release_event can occur in or out of the canvas area. If the event occurs outside of the canvas area the point may contain negative values, or values greater than the canvas area dimensions.

button_press_event

Called when the left mouse button is pressed and the mouse pointer is in the canvas area. The event is passed the point of the button press as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

button_release_event

Called when the left mouse button is released. The event is passed the point of the button release as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values.

The button_release_event can occur in or out of the canvas area. If the event occurs outside of the canvas area the point may contain negative values, or values greater than the canvas area dimensions.

double_click_event

Called when the left mouse button is double-clicked. The event is passed the point of the double-click as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values. The sequence of event calls for a left mouse button double click is:

1. button_press_event
2. button_release_event
3. double_click_event
4. button_release_event

A motion_event is called if the mouse is moved before either button_release_event. A motion_event is called for each mouse movement.

error_event — Called for error with the object.

expose_event — Called when new canvas area is exposed. New canvas area is usually exposed after a horizontal or vertical scroll. The sequence of events for a scroll is:

1. scroll_event
2. expose_event

An expose_event is also called when the dialog box is first displayed. The expose_event is called with the start point, the end point, and a Boolean variable indicating if more expose_event points are coming.

initialize_event — Called before dialog box is displayed, after the dialog box initialize_event.

keyboard_event	Called when keyboard action occurs and the mouse pointer is in the canvas area. The event is passed the ASCII key code, a Boolean value indicating if the SHIFT key is pressed, and a Boolean value indicating if the CONTROL key is pressed.
motion_event	Called when the mouse is moved while the left mouse button is pressed. The event is passed the end point of the motion as a two-element array, and the state of the CONTROL and SHIFT keys as Boolean values. The motion_event can occur in or out of the canvas area. If the event occurs outside of the canvas area the point may contain negative values, or values greater than the canvas area dimensions.
resize_event	Called when the dialog box is resized.
scroll_event	Called when a horizontal or vertical scroll occurs. The array is passed the position of the upper left corner of the canvas after the scroll as a two-element array. An expose_event occurs after a scroll event, unless a fast scroll is generated by dragging a scroll bar. If a fast scroll occurs, multiple scroll_events may be called, followed by multiple expose_events.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.

update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

ComboBoxClass

changed_event	Called when an combo box selection changes. The event is called with the index value of the new choice.
error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

EditBoxClass

button3_menu_state_event

Called before a popup menu is displayed. A popup menu is programmed with the button3_menu_info@ method.

changed_event	Called when the edit box value changes. The method is called when the edit box loses focus. The changed_event is called, then the focus_out_event is called.
error_event	Called for error with the object.
focus_in_event	Called when the edit box receives focus, either by clicking in the edit box or tabbing to the edit box. The focus_in_event is called after the focus_out_event of the control that loses focus.
focus_out_event	Called when the edit box loses focus, either by clicking on a different dialog box control or tabbing out of the edit box.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event. If the edit box has focus when the dialog box is dismissed, the following sequence occurs. 1. If the edit box value has changed, a changed_event is called. 2. A focus_out_event is called. 3. The dialog box terminate_event is called. 4. The edit box terminate_event is called.
time_out_event	Called when the time limit set in the object with the timer@ method expires.

typing_event	Called when the edit box has focus and text is typed into the edit box. The event is called after a character is typed.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

EntryFieldClass

button3_menu_state_event	Called before a popup menu is displayed. A popup menu is programmed with the button3_menu_info@ method.
changed_event	Called when the entry field value changes. The method is called when the entry field loses focus. The changed_event is called, then the focus_out_event is called.
error_event	Called for error with the object.
focus_in_event	Called when the entry field receives focus, either by clicking in the entry field or tabbing to the entry field. The focus_in_event is called after the focus_out_event of the control that loses focus.
focus_out_event	Called when the entry field loses focus, either by clicking on a different dialog box control or tabbing out of the entry field.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.

terminate_event — Called after the dialog box is dismissed, after the dialog box terminate_event. If the entry field has focus when the dialog box is dismissed, the following sequence occurs.

1. If the entry field value has changed, a changed_event is called.
2. A focus_out_event is called.
3. The dialog box terminate_event is called.
4. The entry field terminate_event is called.

time_out_event — Called when the time limit set in the object with the timer@ method expires.

typing_event — Called when the entry field has focus and text is typed into the entry field. The event is called after a character is typed.

update_event — Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

ListBoxClass

button3_menu_state_event

Called before a popup menu is displayed. A popup menu is programmed with the button3_menu_info@ method.

changed_event — Called when a list box selection changes. The event is called with the index value of the new choice.

double_click_event	Called when a double-click on a list box item occurs. The changed_event is called first, then the double_click_event is called.
error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
multi_select_event	Called for multiple selections. Set the is_multi_select@ method to TRUE to allow multiple selections. The changed_event is called for the first selection, the multi_select_event is called for subsequent CTRL-Click selections. The event is also called when an item is de-selected. The changed_event is called if de-selecting a list box item leaves only one item selected. The event receives an array of item indices, in the selection order of the items.
resize_event	Called when the dialog box is resized.
stroke_event	Called when multiple items are selected with a mouse stroke. The event is called when the mouse button is released. A SHIFT-Click (pressing SHIFT and a left mouse button click) key sequence generates a stroke_select_event, selecting list box items from the selected line to the line at the mouse cursor position. The event receives an array of item indices.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.

time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

OptionMenuClass

changed_event	Called when an option menu selection changes. The event is called with the index value of the new choice.
error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

RadioBoxClass

changed_event	Called when a radio box selection changes. The event is called with the index value of the new choice.
error_event	Called for error with the object.

initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

RowColClass

changed_event	Called when a RowCol selection changes. The event is called with the index value of the new choice.
error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

ScalerClass

changed_event	Called when the scale value changes. The event is called when the mouse button is released after a scale drag, or when a mouse click occurs on either side of the scale, in the scale area. The event receives the current value of the scale.
error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
motion_event	Called when the scale is dragged. The event receives the value at the current scale position in the drag. The motion_event is called multiple times during a scale drag.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

SplitterClass

changed_event	Called when the splitter value changes. The event is called when the mouse button is released after a splitter drag. The event receives the current value

of the splitter, the distance between its initial y-position and maximum value, inclusive. The sequence of events for a splitter movement is:

1. picked_event
2. motion_event
3. released_event
4. changed_event

error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
motion_event	Called when the splitter is dragged. The event receives the value at the current splitter position in the drag. The motion_event is called multiple times during a scale drag.
picked_event	Called when the splitter is selected with a mouse button click. The event receives the value of the splitter at the point it is selected.
released_event	Called when the mouse button is released, after the splitter is selected. The event receives the value of the splitter at the point of the mouse button release.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.

update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

TabControlClass

changed_event	Called when an tab selection changes. The event is called with the index value of the new choice.
error_event	Called for error with the object.
initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.
tab_event	Called when a tab layer is selected The tab layer name is passed as an argument.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

TableClass

button3_menu_state_event

Called before a popup menu is displayed. A popup menu is programmed with the button3_menu_info@ method. If menu info is set,

the button3_double_click_event, button3_motion_event, button3_press_event, and button3_release_event are disabled.

cell_changed_event

Called when the contents of a cell change and click in a different cell. The event is called with the row number, the column number, and the new value. The sequence of events when a cell is changed and a click in a different cell are:

1. cell_changed_event
2. cell_focus_out_event

cell_focus_in_event

Called when a click occurs in a table cell. The event receives the row number and column number of the cell.

cell_focus_out_event

Called when a text cursor is in a cell and a click occurs in a different cell. The event receives the row number and column number of the current cell containing the text cursor. The sequence of events when a click occurs in a different cell are:

1. cell_focus_out_event
2. cell_focus_in_event

The cell_focus_in_event is called for the new cell, but the text cursor is not placed in the cell. A second click in the cell calls another cell_focus_in_event and places the text cursor in the new cell.

column_resize_event

Called when a column is resized. The event receives a two-elemnt array containing the column number and the column width. Column numbers are 0-based.

double_click_event

Called when the left mouse button is double-clicked on a row marker. The event is passed the row number. Row numbers are 0-based. The event is called after a selection_changed_event.

If is_multi_select@ is set to TRUE, double-clicking on a row that is already selected calls a selection_changed_event, then a double_click_event. If the CONTROL button is pressed during a double-click, then two selection_changed_events are called.

If is_multi_select@ is set to FALSE, double-clicking on a row that is already selected calls two selection_changed_events.

error_event Called for error with the object.

initialize_event Called before the dialog box is displayed, after the dialog box initialize_event.

request_data_event

Called when a table scroll occurs and the model_is_data_request@ method is set to TRUE. The event receives a two-element array containing the starting row number, and the number of rows.

resize_event Called when the dialog box is resized.

selection_changed_event

Called when row selections change by clicking on the row marker. The event receives the selected row numbers as an array of integers. Row numbers are 0-based.

resize_event — Called when dialog box is resized.

terminate_event — Called after the dialog box is dismissed, after the dialog box terminate_event.

time_out_event — Called when the time limit set in the object with the timer@ method expires.

typing_event — Called when a table cell has focus and text is typed into the cell. The event is called after a character is typed.

update_event — Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

ToggleButtonClass

changed_event — Called when the toggle button value changes. The event receives one of the following values indicating the toggle button state.

- 0 Toggle button is off.
- 1 Toggle button is on.
- 2 Toggle button is grayed, available when is_three_state@ is set to TRUE.

error_event — Called for error with the object.

initialize_event	Called before the dialog box is displayed, after the dialog box initialize_event.
resize_event	Called when the dialog box is resized.
terminate_event	Called after the dialog box is dismissed, after the dialog box terminate_event.
time_out_event	Called when the time limit set in the object with the timer@ method expires.
update_event	Called when the dialog box update@ method is called. The event is first called after the dialog box is displayed.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

S

T

U

V

W

X