# Chapter 1—Omnis External Components

## Introduction

Omnis external components are plug-in modules that extend the range of visual and non-visual objects available in the design and runtime environments in Omnis, as well extending the Omnis programming language. There are many different external components supplied with Omnis, but you can create your own using your own software development tools and the information in this manual.

Once built and installed into Omnis, external components behave in exactly the same way as standard built-in Omnis components. You can change the properties of an external component in design mode using the Property Manager. Likewise, at runtime you can manipulate an external component using methods and the notation, and examine its runtime properties in the Notation Inspector. External components can also contain functions or methods and events, which you can call or intercept using Omnis methods. You can build all of these features into your own external components.

The type and range of external components include:

1. **Window objects (including background objects) and Report objects**  
   Both window and report external components appear in the Omnis Component Store and can be used in your libraries in exactly the same way as built-in GUI objects.
2. **Static Functions**  
   Static functions are components that contain functions, that appear in the Omnis Catalog under the ‘Functions’ group. These functions can be invoked from calculations in your Omnis code.
3. **Omnis objects**  
   Omnis objects or so-called ‘non-visual’ components are objects that can contain methods and properties, which can be used in the Omnis language or called to perform some specific function. External objects can be sub-classed, just like normal Omnis objects, to form new objects. The SQL DAMs are examples of non-visual components.

## Creating your own External Components

Using the libraries supplied, you can create Omnis external components that run under all platforms supported in Omnis. All of the samples supplied, except QuickTime, have independent source code. The Omnis resource compilers for Linux and Mac OSX (Xcode) are supplied. These compile simple Windows style .RC files, and support image types .BMP, allowing the entire component to be portable.

### Components in Omnis

When you start Omnis, you have to tell it to load your new component. You can load an external component via the #EXTCOMP system table. You can access this via the Browser, or open a window class in design mode and right-click on the Component Store, and select the External Components option.

If the components you create are OK, they should appear in the #EXTCOMP system table. If you cannot find your component in the external component list, check the Omnis Trace Log window. Omnis will always write any errors to the trace log during startup. Use one of the radio button options to load the component. Close the dialog. When you return to the design window, you should see your component in the Component Store, under the External Components button in the Component Store toolbar. You should be able to drag the control on to a window class and your component is created.

### Windows and Child Windows

Omnis supports two window types. Top level windows and child windows. In the Omnis IDE, you can create top level windows as window classes, and design the contents of the window by adding controls such as buttons and lists. All window controls such as button and list controls are child windows. A child window is a window that sits inside another parent window. Child windows can also contain other window controls, thus the parent-child relationship can be nested at several levels. For example, a scrollbox window field is a child control within the window class, but it can have other child controls placed within it, thus making it a parent.

An external component operates inside a child window and performs some kind of operation within the child window. The component can do virtually anything from draw a graph, scroll a message, or pick up a click within it and send a message back to Omnis. To do this, the window receives and processes messages. A message informs the child window of all events that affect it, such as the user clicking on it with the mouse. Later, when you create a component, you need to tell Omnis the name of a procedure that Omnis can call with your message. This procedure is often referred to as the WNDPROC (short for Window Procedure) or message handler. There are many messages defined by the component library that your procedure is sent, some you will want to deal with, others you can ignore; you will see how to deal with these messages.

### Data types Defined by the Component Library

To help you write platform-independent components, you should use the data types declared by the component library. All APIs in the library use the following data types.

|  |  |  |
| --- | --- | --- |
| C-type | Omnis type | Description |
| unsigned char or unsigned long\* | qchar | standard unsigned char value \*qchar is defined as 4 bytes for Unicode targets. |
| char or unsigned short | qoschar | platform API-dependent Unicode character. 2 bytes for Win32 & Mac OSX Unicode targets. 1 byte for Linux targets & non-Unicode targets. |
| unsigned char | qbyte | assumed to hold 0-255 |
| unsigned char | qbool | assumed to hold qtrue or qfalse |
| short | qshort | standard short value |
| unsigned short | qushort | standard unsigned short value |
| long | qlong | standard long value |
| unsigned long | qulong | standard unsigned long value |
| platform dependent | qreal | used for real arithmetic |
| short | qret | return type from some API calls |
| enum | qnil | can be used to assign to some objects to clear them |
| unsigned char | qint1 | 1 byte unsigned integer (as stored on disk) |
| short | qint2 | 2 byte integer (as stored on disk) |
| unsigned short | qword2 | 2 byte unsigned integer (as stored on disk) |
| long | qint4 | 4 byte integer (as stored on disk) |
| unsigned long | qword4 | 4 byte unsigned integer (as stored on disk) |
| long | rstrno | uses when calling RESxxx functions |
| short | attnum | property numbers |
| qbool | qfalse = 0 | false boolean value |
| qbool | qtrue = 1 | true boolean value |
| qret | e\_ok = 0 | no error occurred |
| qret | e\_negative = 1 | error occurred |

As well as using the data types, you should try to use the component API as much as possible to ensure platform independent code. In the long run, it may mean you have to recompile for another platform, rather than having to port lots of code.

### Types of visual components

Omnis supports different types of external component which you can add to window and report classes. When Omnis starts up, the component specifies what kind of component it is, and what class type it should appear in.

1. **cObjType\_Basic**  
   a generic window class component.
2. **cObjType\_Picture**  
   a derived Omnis picture component for window classes.
3. **cObjType\_List**  
   a derived Omnis list component for window classes.
4. **cObjType\_DropList**  
   a derived Omnis droplist component for window classes.
5. **cObjType\_IconArray**  
   a derived Omnis icon array component for window classes.
6. **cObjType\_PriOutput**  
   a custom report output device
7. **cRepObjType\_Basic**  
   a generic report class component.
8. **cRepObjType\_Picture**  
   a derived Omnis picture component for report classes.

Components can be both window and report objects. For example, you may want to create a picture-handling component, that works in both window and report mode, therefore its returns type should be:

###### cObjType\_Picture | cRepObjType\_Picture

**Basic Components**

Basic components are generic controls that receive all messages via the WNDPROC. You must code all actions that you want to happen inside your control.

See the examples provided.

**Picture Components**

Picture components are objects derived from the internal Omnis picture field. Omnis calls your WNDPROC with standard messages, but you also receive some specific messages only for derived picture controls. For example, Omnis calls you to inquire how big your image is, so it can handle the scrolling and call you to paint.

See the PCX example.

**List Components**

List components are objects derived from the internal Omnis list field. Omnis calls your WNDPROC with standard messages, but you also receive some specific messages only for derived list controls. For example, when Omnis paints your list, you are called to draw individual lines, possibly in a selected state.

See the PICLIST example.

**Droplist Components**

Droplist components are objects derived from the internal Omnis droplist field. Omnis calls your WNDPROC with standard messages, but you also receive some specific messages only for derived droplist controls. For example, when Omnis paints your droplist contents, you will be called to draw individual lines.

**Icon Array Components**

Icon array components are objects derived from the internal Omnis icon array field. Omnis calls your WNDPROC with standard messages, but you also receive some specific messages only for derived icon array controls. For example, when Omnis paints your icon array, you will be called to draw individual icons and icon labels.

See the ICNARRAY example.

**Report Components**

Report components should be treated as if they were window components. When printing is required, you are called with specific report printing messages.

See the PCX example.

**Background Components**

Background components are objects that behave like internal Omnis background objects. For example, background objects never have the focus or receive events. They are always drawn as part of the background of the window they belong to. One of the sample background components supplied is an object that allows a bitmap to be tiled over an area.

Background components do not have their own **cObjType\_XXX** type, and need to be defined as a **cObjType\_Basic** type component. A flag needs to be set on ECM\_CONNECT to indicate the component should be treated as a background component.

However, it is important to note that you cannot have both background and other visible components in the same library.

See the TILE or WASH example.

### Types of non-visual components

Omnis supports various types of non-visual components. In this context, ‘non-visual’ means a component that does not have a visual interface but one that provides some functionality, such as functions or methods, that can be used in the Omnis programming language. Most non-visual components do not need to be placed on a window or report for their functions or methods to be called. Non-visual and visual components may co-exist in the same library.

**Picture Format Conversion**

Picture format conversion are libraries which provide functionality to convert from the specified format to a native O/S picture and visa-versa.

See PCX example.

**Static Functions**

Static functions behave just like Omnis functions and appear in the catalog just like in-built Omnis functions.

See the FILEOPS example.

**Object Components**

Object components appear in Omnis as objects and can therefore be utilized by adding an ‘Object’ variable (with the appropriate sub-type).

Just like Omnis object classes, external object components may be sub-classed to form new objects.

See the FILEOPS example.

**DAMs**

Writing custom Data Access Modules for Omnis Studio is discussed in the "Omnis Studio DAM API" chapter. This chapter discusses the additional damlib library needed to build these specialised non-visual components as well as datatypes, structures, classes and general techniques involved.

See the GENERIC DAM example.

### Writing Thread-Safe Components

Where several instances of your component may be in use at the same time, you will need to design your code with thread isolation in mind. Use of object-oriented techniques provides the basis for thread-safety as this gives each object instance its own memory and variables.

Where shared memory or commonality exists between multiple instances, the C/C++ programming language facilitates thread management, semaphores and mutual-exclusive execution (MUTEXs) which can be used to control access to the shared resources. Any such commonality should be identified at the design stage.

You can also enhance the thread-safety of your component by passing the EXT\_FLAG\_SESSION flag to Omnis when processing the ECM\_CONNECT message. (See Structures, Messages and Functions for more details).

### Source Files on the Omnis web site

To build an Omnis external component, you must use the latest versions of the following applications:

1. **Microsoft Windows**  
   Microsoft Visual Studio 2015.
2. **macOS**  
   macOS 10.11.5 or later, Xcode 8.2.   
   See Appendix A about creating and porting external components for Mach-O.
3. **Linux**  
   GNU g++ compiler version 4.2.1.   
   All examples ship with makefiles, which can be used with the make utility.

You can download the SOURCE trees for external components from the Omnis website at the following location:

1. [www.omnis.net/products/components/buildyourown.jsp](http://www.omnis.net/products/components/buildyourown.jsp)

There is one source tree for every supported platform. Each source tree contains example external components, ranging from the very simple, GENERIC (lets you create a basic shell component, and previously supplied as a tutorial), to the more complicated controls such as CALENDAR or QuickTime (QuickTime is Windows and Mac only). The source code for the components is generally 100% cross platform and has been duplicated in the various source trees. A few components have some code which is not shared by all platforms. Such platform dependent source will usually be found in source files with names that start with an ‘x’. The source trees have makefiles or projects that can be used to build the components.

When creating external components, try to keep to the tree structure, that is, if you want to create a new component called ‘Simple’, create a directory called Simple inside the source tree. Keeping to the structure will help when porting to other platforms.

For the purpose of this tutorial, rename the source tree to XCOMP.

### Getting Started with Generic

One of the many samples supplied to help you create Omnis external components is **generic**. There are four versions of this control in the source tree that explain how to write Omnis components and give you a useful starting framework for building your own components. Before you begin to write some code, you need to setup your development environment.

### Setting up your development tree

Inside the source tree you will find the following folders which are of special importance.

COMPLIB – Header files and libraries for building XCOMP and WEBCOMP components.

EXTLIB – Header files for building OLD TYPE externals.

HEADERS - Header files for building web client components

LIBS - Libraries for building web client components (win32 and linux only)

JPEGLIB - Libraries for building HTML device

ORFCSTAT - Library for building web client components

On Mac OS, there are some additional folders which are of special importance (we will bring the other platforms in line in future releases). The Mac OS projects we ship place the components they build into the following folders:

\_OSXUnicode - release versions of XCOMP components  
\_OSXUnicodeDbg - debugging versions of XCOMP components  
\_OSXUnicodeWeb - release versions of WEB CLIENT components  
\_OSXUnicodeWebDbg - debugging versions of WEB CLIENT components  
\_OSXUnicodeWebDesign - release versions of WEBCOMP components  
\_OSXUnicodeWebDesignDbg - debugging versions of WEBCOMP components

**Mac OS:**

For the Mac Xcode environment we also supply various stationery and a resource compiler that you will need to install.

* The macOS SDK should be installed with Xcode: your component needs to use the macOS 10.12 SDK with a minimum deployment target of 10.9
* Copy the Mach-O resource compiler ( omnisrc64.app ) into /Applications/Xcode.app/Contents/Developer/Tools

This is the new Omnis Resource Compiler and is included in the tools folder supplied with this document.

For further information on building components for Mac OSX 10.5 and later, please refer to Appendix A.

**Linux OS:**

For the Linux environment you will need to set a few environment variables and configure the resource compiler.

Please note that these instructions assume that you installed the source tree in ‘/’ resulting in a source tree called ‘omnisext’. If you installed the tree elsewhere then you will need to change ‘omnisext’ with your installation path.

* Set the environment variable **LD\_LIBRARY\_PATH** by typing :

###### LD\_LIBRARY\_PATH=/omnisext/omnisxi:$LD\_LIBRARY\_PATH export LD\_LIBRARY\_PATH

* Set the environment variable **V4DIR** by typing :

###### V4DIR=/omnisext/omnisxi export V4DIR

* Configure the resource compiler by setting the environment variable **OMNISRC** by typing :

###### OMNISRC=/omnisext/omnisrc export OMNISRC

* If you installed to a folder other than ‘/’, you will need to edit the **omnisrc.ini** by typing:

###### emacs /omnisext/omnisrc/omnisrc.ini

The two sub-sections **Template** and **IncludeDirs** contained within the section **[Setup]** needs to be modified to point to the installation folder.

The defaults values are :-

###### Template=/omnisext/omnisrc/omnisrc.tpl IncludeDirs=/omnisext/omnisxi;/omnisext/complib;/omnisext/extlib

* When you are satisfied with your changes, choose ‘Save’ and exit
* Copy the **omnisrc** executable to a folder, which is on your search path  
  For example: cp /omnisext/omnisrc/omnisrc /usr/bin/omnisrc

**All platforms**

When you have built your component, you should add your Windows DLL or Mac OS Code Fragment to the XCOMP or WEBCOMP folder located inside the main Omnis folder, or to your web client installation, depending on what you are building. Run Omnis and use the #EXTCOMP system table to load the component. If all is well, the control appears in the Component Store and can be dragged on to a window or report class. Any load errors are reported in the Omnis trace log.

Now you have setup your build environment and tree we can get to work creating our new component.

**If you are using Windows:**

* Startup Visual Studio
* Go to the File menu and select New / Project
* From the Visual C++ project types select ATL, enter a name and click OK
* Click Finish

**If you are using Mac OS X:**

* Startup Xcode
* Go to the File menu and choose the New Project... option
* Select 'Empty Project' and click Next
* Enter a project name and click Finish

**If you are using Linux:**

* Create a new folder in the **/omnisext** folder
* Copy the file **extcomp.mak** from **/omnisext/generic** into your new folder. This file is a generic makefile which is the same for all of the examples (except HTML)
* Copy the file **makefile** from /**omnisext/generic** into your new folder. This file contains the name of the component and the source files needed to build it

**Generic.cpp**

You are now ready to create the generic external component. Create a new file called ***generic.cpp*** and enter the following:

###### #include <extcomp.he> #include <hwnd.he> #include <gdi.he> #include "generic.he"

###### extern "C" qlong OmnisWNDPROC GenericWndProc(HWND hwnd, LPARAM Msg, WPARAM wParam, LPARAM lParam, EXTCompInfo\* eci) {   ECOsetupCallbacks(hwnd, eci);   switch (Msg)   {     case ECM\_GETCOMPLIBINFO:     {       return ECOreturnCompInfo( gInstLib, eci, LIB\_RES\_NAME, OBJECT\_COUNT );     }

###### case ECM\_GETCOMPID:     {       if ( wParam==1 )         return ECOreturnCompID( gInstLib, eci, OBJECT\_ID1, cObjType\_Basic );       return 0L;     }

###### case ECM\_GETCOMPICON:     {       if ( eci->mCompId==OBJECT\_ID1 )         return ECOreturnIcon( gInstLib, eci, GENERIC\_ICON );       return qfalse;     }

###### case ECM\_OBJCONSTRUCT:     {       tqfGenericObject\* object = new tqfGenericObject( hwnd );       ECOinsertObject( eci, hwnd, (void\*)object );       return qtrue;     }

###### case ECM\_OBJDESTRUCT:     {       tqfGenericObject\* object = (tqfGenericObject\*)ECOremoveObject( eci, hwnd );       if ( NULL!=object )       {         delete object;       }       return qtrue;     }   }   return WNDdefWindowProc(hwnd,Msg,wParam,lParam,eci); }

Let’s look at this in more detail. First the includes:

###### #include <extcomp.he> #include <hwnd.he> #include <gdi.he> #include "generic.he"

***extcomp.he***, ***hwnd.he*** *and* ***gdi.he*** are external component library header files. ***extcomp.he*** declares various external component specific APIs; ***hwnd.he*** declares the child window API calls; ***gdi.he*** declares the graphical API calls; and ***generic.he*** which you will create below.

The message procedure is as follows.

###### extern "C" qlong OmnisWNDPROC GenericWndProc(HWND hwnd, LPARAM Msg, WPARAM wParam, LPARAM lParam, EXTCompInfo\* eci)

**OmnisWNDPROC** is a #define that the Omnis component library has setup. This defines some calling conventions that vary from platform to platform. For now this is all you really need to know. Next is the name, GenericWndProc. This is the message procedure Omnis calls with your child window messages. This procedure has the following parameters:

|  |  |  |
| --- | --- | --- |
|  |  |  |
| HWND | hwnd | This is a handle to the child window the message is for |
| LPARAM | Msg | This is the message |
| WPARAM | wParam | This is extra information for the message |
| LPARAM | lParam | This is extra information for the message |
| EXTcompInfo\* | eci | This is a pointer to some information about your component |

When Omnis calls the message procedure, it sends along the HWND in the **hwnd** parameter. This is the child window the message was for. Complex components may have many child windows all using the same message procedure. Using the HWND helps the component do the right thing for the right child window.

**Msg** is the message. There are many messages that can be sent to your procedure, such as WM\_PAINT or WM\_LBUTTONDOWN.

**wParam** is some extra information for the message. Sometimes messages need to pass other information, such as the cursor position when the WM\_LBUTTONDOWN was generated.

**lParam**, like wParam, is used for extra message information.

**eci** is a pointer to a structure holding information about your component. It is used with various API calls. See later.

Next is the first and *most important* line of the message procedure.

###### ECOsetupCallbacks(hwnd, eci);

Most of the API calls call Omnis for some information, or to do some processing. This line enables the call to Omnis to work. If this line is missing, your component will crash.

Next there is a switch statement testing the message parameter:

***ECM\_GETCOMPLIBINFO.*** This is the first message the message procedure handles. Omnis is calling your message procedure trying to find out how many controls your component supports, and the name of your library.

###### return ECOreturnCompInfo( gInstLib, eci, LIB\_RES\_NAME, OBJECT\_COUNT );

Here you return the result of a function call *ECOreturnCompInfo*.This function is described later, but usually takes a string resource number which holds the name of your component library, and takes the number of controls your component contains.

***ECM\_GETCOMPID***. Omnis now knows how many controls you are intending to support in your component library due to the result of the last message. It now wants to know what ID each control within the component library should have. The id can be any number you decide to associate with the control. In the future when Omnis wants something to happen to a control, it uses the id you return here. Omnis calls you with this message *for 1 to n* times, where n is the number of controls your library supports. The calling count is passed in ***wParam***.

###### if ( wParam==1 )   return ECOreturnCompID( gInstLib, eci, OBJECT\_ID1, cObjType\_Basic ); return 0L;

Since generic supports one control (**OBJECT\_COUNT=1,** this is defined in your header file), you wait for a call where wParam is 1. On this message, you return the result of ***ECOreturnCompID***. This API specifies the controls id, and the type of control you want it to be. See ***Types Of Component*** later in this document. Here you indicate the control has an id of **OBJECT\_ID1** and is a **cObjType\_Basic** basic component.

***ECM\_GETCOMPICON***. Now Omnis knows how many controls your library has and the id for each control, it asks for the icon to use in the Omnis Component Store.

###### if ( eci->mCompId==OBJECT\_ID1 )   return ECOreturnIcon( gInstLib, eci, GENERIC\_ICON );

###### return qfalse;

Here, you are checking a member of the **eci** parameter, **mCompId.** This is set to an id you returned from the last message (**OBJECT\_ID1)**. The ***ECOreturnIcon*** API is described later, but generally it extracts a **.bmp** ( bitmap ) from the resource file so you can return it to Omnis.

With regards to setting up your component so that Omnis knows it is there, these messages are generally all you need. The next set of messages are used when you place your component on a window or report class. When that happens, Omnis calls your message procedure with many more messages. Here are the important ones.

***ECM\_OBJCONSTRUCT***. Omnis is calling the message procedure as it is just about to create an instance of your object. This can happen when you drag a component out of the Component Store on to a design window or report class, or a window class is being opened in runtime mode, or a report is being printed. This is the code you need to execute:

###### tqfGenericObject\* object = new tqfGenericObject( hwnd ); ECOinsertObject( eci, hwnd, (void\*)object ); return qtrue;

The first line creates a new object called ***tqfGenericObject***, which is defined below. This class performs all of the operations for your control. Next it is calling ***ECOinsertObject***. This API adds a pointer to the ***tqfGenericObject*** just created into a chain of objects. The pointer and the ***hwnd*** being passed are stored in the chain. The external component library maintains this chain, so later when a message arrives in your message procedure, you can ask for the object ( ***tqfGenericObject*** ) based on the child window the message was for, and get the correct object to process the message. This is necessary as multiple instances of your component can be created.

Finally you return qtrue. This informs Omnis you have processed the message. Not all messages expect qtrue to indicate the message was handled. The return value from all messages can be found in this manual.

***ECM\_OBJDESTRUCT*** is the next message. Omnis is calling the message procedure as it is just about to delete an instance of your object. This can happen when you close a window class containing your component, or a report has finished printing your component:

###### tqfGenericObject\* object = tqfGenericObject\*)ECOremoveObject(eci,hwnd); if ( NULL!=object ) {   delete object;

###### }

###### return qtrue;

The first line here is calling ***ECOremoveObject*** to find an object in the chain of objects based on the passed **hwnd.**If an object is found, it is removed from the list and a pointer to the object is returned. If the pointer is valid, you delete it, freeing all memory previously allocated. Again, you return qtrue to inform Omnis you have processed the message.

Finally:

###### return WNDdefWindowProc(hwnd,Msg,wParam,lParam,eci);

This is another *very*important line. Remember that many messages are sent to the message procedure, some important, others not so important. This is where the not so important messages should go. ***WNDdefWindowProc*** is an API to which all messages not handled should be passed. This allows Omnis to do the default operation for messages you do not want to handle.

To complete this file, enter the following:

###### tqfGenericObject::tqfGenericObject( HWND pFieldHWnd ) {   mHWnd = pFieldHWnd; }

###### tqfGenericObject::~tqfGenericObject() { }

###### qbool tqfGenericObject::paint() {   WNDpaintStruct paintStruct;   WNDbeginPaint( mHWnd, &paintStruct );   WNDendPaint( mHWnd, &paintStruct );   return qtrue; }

On previous messages, ***ECM\_OBJCONSTRUCT*** and ***ECM\_OBJDESTRUCT*** referred to the ***tqfGenericObject*** class. This class contains the code which makes the component actually do something. You can add to this later.

**Generic.he**

New file, ***generic.he***, enter the following code. This defines the class referred to above.

###### /\* Number of controls within library \*/ #define OBJECT\_COUNT 1 /\* Resource id of library name \*/ #define LIB\_RES\_NAME 1000 /\* Resource id of control within library \*/ #define OBJECT\_ID1 2000 /\* Resource bitmap id \*/ #define GENERIC\_ICON 1

###### class tqfGenericObject {   private:     HWND     mHWnd;

###### public:     tqfGenericObject( HWND pFieldHWnd );     ~tqfGenericObject();     qbool paint(); };

**Generic.rc**

New file, ***generic.rc***, this is the resource file. It is laid out like a Windows (Window OS) resource file. Under Mac OS and Linux, you can use a resource compiler supplied on the Omnis CD which supports a very basic set of resource keywords. If you keep the resource files simple, they will be cross-platform.

###### 1 BITMAP DISCARDABLE "GENERIC.BMP" STRINGTABLE DISCARDABLE

###### BEGIN  1000    "Generic Library"  2000    "Generic Control"  31000   "GenericWndProc" END

The resource file first includes a bitmap ( generic.bmp ). You can either create a small Window **.BMP** file ( 16x16 preferably ), or take a copy of the generic.bmp file from the Omnis CD.

String 31000 is **very important**, as this is the name of the message procedure that Omnis tries to call. If you do not have this string, the name of you message procedure should be ‘OmnisEXTCOMPONENT’. If the message procedure is not called this, and you do not have a string defining the name to call, Omnis will not call you.

**Omnis Web Client**

If you intend to release your component as a web component that can be used with the Omnis Web Client, do not use string number 31020. This is reserved for version number checking.

**.DEF and .EXP files**

Under Windows, an export file is needed for the compiler. This file describes what functions can be called from outside the component. As described above, Omnis needs to call your message procedure with messages for your child windows. For it to do that, the message procedures must be *exported*.

**Under Windows, Generic.def:**

Create a Generic.def file and enter the following:

###### LIBRARY      GENERIC EXPORTS     GenericWndProc @1

This allows Omnis to get and call the message procedure.

**Note for all platforms:** The name entered in string 31000, the name of the message procedure in ***generic.cpp*** and the name in ***generic.def*** must all be identical. If you do not supply a 31000 string, the message procedure name in ***generic.cpp*** and defined in the ***generic.def*** must be ‘OmnisEXTCOMPONENT’.

#### Building the Generic Component

**If you are using Windows:**

* Add the generic.cpp, generic.def and generic.rc files to the project using the Project menu and choose the Add New Item... option.
* Open the project Properties and select the C/C++, preprocessor category from the treelist. Add iswin32, isXCOMPLIB to the preprocessor definitions.
* Select the Linker, Input category from the treelist and add omnisu.lib to the Additional Dependencies.
* Close the Property Pages.
* Go to the Build menu and select Build Solution.

If all is successful, you should have created a generic.dll file. This can be moved to the XCOMP folder of your Omnis installation.

**If you are using Mac OSX:**

* Add the generic.cpp by right-clicking on *Source* and selecting Add→Existing Files. Add generic.rc by right-clicking on *Resources* and selecting Add→Existing Files.
* Select the target and configuration you wish to build and select Build from the Build menu.

If you selected the target UnicodeCore, the component, once built, will be placed in the folder \_OSXUnicode inside the source tree. Please note: The ReleaseBuild targets contain no debugging information.

**If you are using Linux:**

* From the component folder type:   
  - ‘**make Release**’ to build the XCOMP component. The resulting component will be in the releaseuni folder.   
  - ‘**make ReleaseWeb**’ to build the Web Client component. The resulting component will be in the releaseuniweb folder.   
  - ‘**make ReleaseWebDesign**’ to build the WEBCOMP component. The resulting component will be in the releaseuniwebdesign folder.   
  You can substitute Release for Debug if you wish to build debug versions.

**For all platforms:**

If all is successful, you should have created a generic file. You can move it to the XCOMP folder in the main Omnis folder. Remember if Omnis is currently running, you will need to quit and restart.

### Moving On From Generic

The Generic example component you have created is only a shell. It can be dragged from the Component Store and created in runtime mode. Next, you can add a property to the component that will appear in the Property Manager. You can modify component properties in design mode and runtime using the Property Manager.

First you need to add some functionality to the tqfGenericObject class. In ***generic.he*** enter the following:

###### qcol mMyColor; // and qlong attributeSupport( LPARAM pMessage, WPARAM wParam, LPARAM lParam, EXTCompInfo\* eci );

###### // Your class header should now look like this: class tqfGenericObject {   private:     HWND mHWnd;     qcol mMyColor;

###### public:     tqfGenericObject( HWND pFieldHWnd );     ~tqfGenericObject();     qbool paint();     qlong attributeSupport( LPARAM pMessage, WPARAM wParam, LPARAM lParam, EXTCompInfo\* eci ); };

Here a new member is added to the class **mMyColor**. Its type is **qcol**. This is a type defined in **gdi.he** and represents a color value (RGB).

A new member function **attributeSupport** is also added. This function is used when Omnis is doing something with your properties.

Now open ***generic.cpp.***

Go to the tqfGenericObject:: tqfGenericObject method ( constructor ). Add the following line:

###### mMyColor = GDI\_COLOR\_WINDOW;

Go to the tqfGenericObject::paint() method. This was added previously, but until now was unused. Alter the method so it looks like this:

###### qbool tqfGenericObject::paint() {   WNDpaintStruct paintStruct;   WNDbeginPaint( mHWnd, &paintStruct );

###### qrect cRect;   WNDgetClientRect( mHWnd, &cRect );   HBRUSH brush = GDIgetStockBrush( BLACK\_BRUSH );   GDIsetTextColor( paintStruct.hdc, mMyColor );   GDIfillRect( paintStruct.hdc, &cRect, brush );

###### WNDendPaint( mHWnd, &paintStruct );   return qtrue; }

The paint method uses some API calls from both ***hwnd.he*** and ***gdi.he.*** When this method is called as a result of a message, it fills your component with the color that is stored in the new color member **mMyColor.** You should read the HWND and GDI document for an explanation of the APIs used, but generally, the code gets the size of your child window (left, top, width, height), and gets a solid brush. It sets the color of the solid brush to the color in the new color member and then fills the child window with that color.

Back to the message procedure now, and add cases for the following messages:

###### case WM\_PAINT: {   tqfGenericObject\* object = (tqfGenericObject\*)ECOfindObject( eci, hwnd );   if ( NULL!=object && object->paint() )   return qtrue;   break; }

###### case ECM\_GETPROPNAME: {   return ECOreturnProperties( gInstLib, eci, &MyProperties[0], 1 ); }

###### case ECM\_PROPERTYCANASSIGN: case ECM\_SETPROPERTY: case ECM\_GETPROPERTY: {   tqfGenericObject\* object = (tqfGenericObject\*)ECOfindObject( eci, hwnd );   if ( object )     return object->attributeSupport( Msg, wParam, lParam, eci );   return 0L; }

Consider the following messages:

**WM\_PAINT** message informs use the **hwnd** needs painting.

**ECM\_GETPROPNAME** is sent by Omnis to ask for the component’s property table.

**ECM\_PROPERTYCANASSIGN** is sent by Omnis to see if a property can have values assigned.

**ECM\_SETPROPERTY** is sent by Omnis to get the value of a property.

**ECM\_GETPROPERTY** is sent by Omnis to set the value of a property.

When you get a **WM\_PAINT** message, you find the object in the chain of object instances from the hwnd coming into the message procedure. If you find the object, you call the ::paint() member function of the object.

When you get a **ECM\_GETPROPNAME** message, you call another ECO API to build a property table and return it to Omnis. This API is described later, see *Component Properties*.

Now to add some more code. At the top of the file add the following:

###### const cMyColorProp = 1;

###### ECOproperty MyProperties[] = {   cMyColorProp, 4000, fftInteger, EXTD\_FLAG\_PWINDCOL, 0, 0, 0 };

This table defines your properties. The layout of the table is defined in the *Component Properties*section, but generally it describes the property id, the resource name of the property, its data type, and the type of data as shown in the Property Manager. The property table, when returned to Omnis using the code shown below, controls how your properties are handled.

###### return ECOreturnProperties( gInstLib, eci, &MyProperties[0], 1 );

The only thing left to do in this file is to add the ::attributeSupport() method you declared in the header file. Somewhere near the tqfGenericObject class add the following:

###### qlong tqfGenericObject::attributeSupport( LPARAM pMessage, WPARAM wParam, LPARAM lParam, EXTCompInfo\* eci ) {   switch( pMessage )   {     case ECM\_PROPERTYCANASSIGN:     {       return 1L;     }

###### case ECM\_SETPROPERTY:     {      EXTParamInfo\* param = ECOfindParamNum( eci, 1 );       if ( param )      {         EXTfldval fval( (qfldval)param->mData );        switch( ECOgetId(eci) )         {           case cMyColorProp:           {              mMyColor = (qcol)fval.getLong();              WNDinvalidateRect( mHWnd, NULL );              break;            }         }       }       return 1L;     }

###### case ECM\_GETPROPERTY:     {       EXTfldval fval;       switch( ECOgetId(eci) )       {         case cMyColorProp:         {           fval.setLong( (qlong)mMyColor );           break;         }       }       ECOaddParam(eci,&fval);       return 1L;     }   }   // no property found or message was wrong   return 0L; }

This method is called when Omnis needs to do something with your properties. This is covered in more detail within the *Component Properties* section later, but generally it lets you handle your color property or any future properties you decide to add. For this example, when a color property is assigned, you alter the member in the **tqfGenericObject** class with the new color value being sent from Omnis, force your child window to be repainted, resulting in the new color being drawn on screen. When the Property Manager needs to know what the color is, you send it the value back, and you also tell the Property Manager if it is allowed to assign color to your object.

Finally open ***generic.rc***  and add the following:

###### 4000 "$mycolor:This is a color property"

Your RC file should look like this:

###### 1 BITMAP DISCARDABLE "GENERIC.BMP" STRINGTABLE DISCARDABLE

###### BEGIN    1000      "Generic Library"    2000      "Generic Control"    4000      "$mycolor:This is a color property"    31000     "GenericWndProc" END

Now recompile the component. Close Omnis if it is still running, and move the component into your XCOMP folder. Restart Omnis. In the Omnis IDE, when you open a window class and click on your component control, a Custom tab is displayed in the Property Manager. Select it and you should see your color property. Now try assigning some values and it should change the color of your component.

You have covered the very basics of building your own external component. The source contains further generic samples that build on from the basic one adding more properties, events, and component methods.

If you are ready for more of a challenge, the source has many other controls that demonstrate much more of the external components interface. All of the samples supplied (except QuickTime) are completely cross-platform.

Bear in mind Omnis is a cross-platform development tool, and the external component interface has been designed with this in mind. If you want your controls to run on all platforms supported in Omnis, try to use the Omnis API as much as possible. There is very little it cannot do, and if you *only* use the API, your code should remain completely portable, all you need to do is recompile.

### General Hints

Here are some general points you should remember when writing Omnis components:

* Keep to the External Component API; this helps you port your controls to other platforms
* Initialize all members used to handle properties. When the control is first created, the initial values, as displayed in the Property Manager, are the values you initialize your members to.
* Read the ‘Memory Issues’ for the EXTfldval and EXTqlist classes later defined.
* Do NOT nest painting. See ***WNDstartDraw*** and ***WNDendDraw*** in the HWND documentation.
* For large amounts of data, such as picture components, you can use the ***MEMincAddr*** and ***MEMdecAddr*** function to handle large images.
* If you have any problems with your component when you are within the Omnis IDE, such as the DLL not appearing in the #EXTCOMP dialog, check the Omnis trace log. Any problems encountered in Omnis with respect to your components are reported to the trace log.
* String resource 31020 should not be used as it is reserved for Web Client version number checking.
* If you declare a date property ( fftDate ), depending on the date subtype used with the EXTfldval::setDate() API, the Property Manager uses either #FT or #FDT to format the property value.

###### EXTfldval fval; fval.setDate( myDateValue, dpFtime );

This example stores a date value in an EXTfldval object. The Property Manager would use #FT to format value because the date subtype used was dpFtime.

#### Omnis and Microsoft Foundation Classes (Windows Only)

This section describes how to use the Omnis component classes within a Microsoft Foundation Class (MFC) dynamic-linked library.

To use MFC and Omnis classes in a DLL you must include **OmnisMFC.LIB** in the project instead of **Omnis.LIB.** The differences between these two libraries are:

1. The function **DllMain** (Win32) or **LibMain** (Win16) does not exist in the OmnisMFC.LIB library.
2. The global variable **gInstLib** (previously initialized during DllMain or LibMain) does not exist in the OmnisMFC.LIB library.

#### Mac OSX and XCode Resource Files

Please note that Xcode and its underlying build scripts may encounter problems where file or folder names contain spaces. For this reason it is best to use underscores in place of spaces. Alternatively, it may be possible to work around the issue by adding double quotes around various build attributes, for example:

* The two arguments to the *cp* command that follows the *omnisrc* command in the rule for compiling rc files.
* The header search path for the project headers.
* The framework search path.

#### Linux Compilation Issues

Please ensure that your Linux system has the necessary link libraries and development packages installed in addition to the gcc compiler (version 4.1 or higher).

You can obtain the version number of your compiler by typing:

###### gcc –dumpversion

You can install the missing standard C library (under Ubuntu for instance) using the command:

###### sudo apt-get install libstdc++2.10-glibc2.2

## Creating Non-Visual Components

Non-visual components are component libraries which contain either Omnis static functions and/or Omnis external class objects.

Just like in high-level languages such as C++ static functions are useful when processing single non-related tasks. However when functions are related, it is sometimes useful to build a collection of related functions into a class object.

### Static Functions

Static functions are functions which can be used in the Omnis script language in calculations. Component library static functions appear in the ‘Functions’ category in the Catalog window.

Adding static functions to your component library requires the following steps:

1. Add the flag EXT\_FLAG\_NVOBJECTS to the set of flags returned by ECM\_CONNECT message. Without adding this flag, Omnis will not request the list of static functions from your library.
2. Return a list via ECOreturnMethods in response to a ECM\_GETSTATICOBJECT message. This message is sent to the component library when Omnis requires a list of static functions.
3. Respond to ECM\_METHODCALL. However, as these are static functions, the HWND parameter will be NULL. As will wParam and lParam.
4. An example of static functions in use would be (excerpts from FILEOPS):

###### ECOmethodEvent fileStaticFuncs[ cSMethod\_Count ] = { // Unique external ID Resource Number Flags   cSMethodId\_CreateDir , 8000, 0, 0, 0, 0, 0,   ….   cSMethodId\_Rename , 8014, 0, 0, 0, 0, 0 };

###### extern "C" qlong OmnisWNDPROC GenericWndProc(HWND hwnd, LPARAM Msg, WPARAM wParam, LPARAM lParam, EXTCompInfo\* eci) {   ECOsetupCallbacks(hwnd, eci);   switch (Msg)   {     case ECM\_CONNECT:     { // Component library contains NV objects & should always be loaded       return EXT\_FLAG\_LOADED|EXT\_FLAG\_ALWAYS\_USABLE|EXT\_FLAG\_NVOBJECTS;     }     case ECM\_GETSTATICOBJECT:     { // Omnis is requesting a list of our static functions       return ECOreturnMethods( gInstLib, eci, &fileStaticFuncs[0], cSMethod\_Count );     }

###### case ECM\_METHODCALL:     { // Omnis requires a static method to be called       switch ( ECOgetId(pEci) )       {         case cSMethodId\_CreateDir:... Processing          …         case cSMethodId\_Rename: … Processing       }       return 1L;     }   }   return DefWindowProc( hwnd,Msg,wParam,lParam,eci); }

**See also** ECM\_CONNECT, ECM\_GETSTATICOBJECT, ECM\_METHODCALL, ECOreturnMethods

### Class Objects

Class objects are, as in the Omnis language, objects which group together data and functions into a single entity.

Adding class objects requires the following steps :-

* Add the flag EXT\_FLAG\_NVOBJECTS to the set of flags returned by ECM\_CONNECT message. Without adding this flag, Omnis will not request of list of objects from your library.
* Respond to the ECM\_GETOBJECT message and return a list of objects via ECOreturnObjects. It is important to note that your ECOobject structure should contain unique ids. During subsequent calls the EXTCompInfo mCompId member will contain this id to inform you of the type of object.
* Respond to ECM\_OBJCONSTRUCT, ECM\_OBJDESTRUCT and ECM\_OBJECT\_COPY to ensure that your objects are created, destructed and copied.
* Respond to ECM\_GETMETHODNAME and ECM\_GETPROPNAME to return any methods and properties that your object may have.
* Respond to ECM\_PROPERTYCANASSIGN, ECM\_SETPROPERTY, ECM\_GETPROPERTY in normal manual to manage your objects’ properties.
* Finally, respond to ECM\_METHODCALL to inform any of your objects’ methods.

It is important to note that during all of the above messages (except ECM\_GETMETHODNAME, ECM\_GETPROPNAME and ECM\_OBJECT\_COPY) lParam will contain a unique reference to your object. You should use ECOfindNVObject to retrieve your objects’ data.

It is also important to note how you require your objects to be managed. For example the FILEOPS example uses a container to hold the actual object. The object is only released/freed when a reference count gets to zero. This allows several Omnis object variables to point to the same FILEOPS object (similar to COM).

An example of use may be (excerpts from FILEOPS):

###### ECOobject fileObjects[ cObject\_Count ] = { // Unique external ID Resource Number Flags   cObject\_FileOps, 2000, 0, 0 };

###### extern "C" qlong OmnisWNDPROC GenericWndProc(HWND hwnd, LPARAM Msg, WPARAM wParam, LPARAM lParam, EXTCompInfo\* eci) {   ECOsetupCallbacks(hwnd, eci);   switch (Msg)   {     case ECM\_CONNECT:     **{** // Component library contains NV objects & should always be loaded       return EXT\_FLAG\_LOADED | EXT\_FLAG\_ALWAYS\_USABLE | EXT\_FLAG\_NVOBJECTS;     }

###### case ECM\_GETOBJECT:     **{** // Omnis is requesting a list of our object class names return ECOreturnObjects(gInstLib,eci, &fileObjects[0], cObject\_Count);     }

###### case ECM\_OBJCONSTRUCT:     **{** // Omnis is requesting the construct of a new object.      if ( eci->mCompId==cObject\_FileOps )       {        tqfFileOpsContainer\* object = (tqfFileOpsContainer\*) ECOfindNVObject(eci->mOmnisInstance, lParam );        if ( !object )         {           tqfFileOpsContainer\* obj = new tqfFileOpsContainer((qobjinst)lParam);           ECOinsertNVObject(eci->mOmnisInstance,lParam,(void\*)obj);         }         return qtrue;       }       return qfalse;     }

###### case ECM\_OBJDESTRUCT:     **{** // Omnis is requesting the destruction of your object       if (eci->mCompId==cObject\_FileOps && wParam==ECM\_WPARAM\_OBJINFO)       {         void\* object=ECOremoveNVObject(eci->mOmnisInstance,lParam );         if ( object )         {           tqfFileOpsContainer\* fileOps = (tqfFileOpsContainer\*)object;           delete fileOps;         }       }       return qtrue;     }

###### case ECM\_OBJECT\_COPY:     **{** // Omnis requires a new object to be created from an existing one       objCopyInfo\* copyInfo = (objCopyInfo\*)lParam;       tqfFileOpsContainer\* srcobj = (tqfFileOpsContainer\*) ECOfindNVObject(eci->mOmnisInstance,copyInfo->mSourceObject);       if ( srcobj )       {         tqfFileOpsContainer\* destObj = qfFileOpsContainer\*)ECOfindNVObject(eci->mOmnisInstance, copyInfo->mDestinationObject );         if ( !destObj )         {           destObj = new tqfFileOpsContainer( (qobjinst)copyInfo->mDestinationObject,srcobj);           ECOinsertNVObject( eci->mOmnisInstance, copyInfo->mDestinationObject, (void\*)destObj );         }         else           destObj->setObject( (qobjinst)copyInfo->mDestinationObject,srcobj );       }       break;     }

###### case ECM\_GETMETHODNAME:     **{** // Omnis is requesting a list of our objects’ methods       if ( eci->mCompId==cObject\_FileOps )         return ECOreturnMethods( gInstLib, eci, &fileObjFuncs[0], cIMethod\_Count );       break;     }

###### case ECM\_GETPROPNAME:     **{** // Omnis is requesting a list of our objects’ properties       // but we don’t have any so simply return      break;     }

###### case ECM\_PROPERTYCANASSIGN:     case ECM\_SETPROPERTY:     case ECM\_GETPROPERTY:     **{**   // Omnis requires property management         // but we don’t have any so simply return         break;     }

###### case ECM\_METHODCALL:     { // Omnis requires a method to be invoked       if ( eci->mCompId==cObject\_FileOps )       {         void\* object = (void\*)ECOfindNVObject( eci->mOmnisInstance, lParam );         tqfFileOpsContainer\* fileOps = (tqfFileOpsContainer\*)object;         if ( fileOps->mObject )           return fileOps->mObject->methodCall( Msg, wParam, lParam, eci );         }         break;       }     }     return DefWindowProc( hwnd,Msg,wParam,lParam,eci);   }

### Control Handlers

This section describes how to develop a control handler component. Control handlers are essentially components which handle other components, such as an ActiveX.

To create a control handler, you should follow these steps. Note that the CONTROLLIB and CONTROL classes are used for illustration purposes and do not exist in the Omnis component environment.

* Add EXT\_FLAG\_CTRLHANDLER to the component flags.
* Process ECM\_GETHANDLERICON to inform Omnis of the HBITMAP to use for the components' group in the Component Store.
* Support for ECM\_GETCOMPLIBINFO must be restructured. The component must provide the information for all the control libraries that it supports. The control library names that the component supports must include a file path as a prefix. An example of this would be:

###### if ( !pEci->mCompLibId ) {   // Omnis is inquiring on the handler.   ECOreturnCompInfo(gInstLib,pEci,CTRL\_RES\_NAME,0);   // Id of first library   pEci->mCompLibId = 1;   return qtrue; } else {   // Omnis is inquiring on a control libraries   CONTROLLIB\* prevLib = getLib( pEci->mCompLibId );   // Find library from id   CONTROLLIB\* nextLib=0;   if ( prevLib )     nextLib=prevLib->mNextLibrary;   if ( nextLib )   {     // you have another library for Omnis     EXTfldval exfldval;     // Format of returned name <FilePath>+'\0'+<Library Name>     str255 ctrlInfo = nextLib->mLibraryPath;     // Space for NULL     ctrlInfo[0]++;     // Terminate Cstring     ctrlInfo[ ctrlInfo.Length() ] = '\0';     // Add control library name     ctrlInfo.concat( nextLib->mLibraryName );     exfldval.setChar( ctrlInfo );     ECOaddParam(pEci,&exfldval);

###### // Set Unique id of this library. The id may change between sessions.     pEci->mCompLibId = nextLib->mLibraryId;

###### // Return number of controls within library      return nextLib->mControlCount;   }

###### // No more libraries   return qfalse; }

* Support for ECM\_GETCOMPID message must return the unique identifier for the control. It should be noted that like the controls' library (mCompLibId), it is not necessary to maintain the same unique identifier between Omnis sessions. Example of ECM\_GETCOMPID:

###### CONTROLLIB\* curLib = getLib( pEci->mCompLibId ); if ( curLib ) {   EXTfldval exfldval;   exfldval.setChar( curLib->getControlName(wParam) );   ECOaddParam(eci,&exfldval);   pEci->mCompId = curLib->getControlId( wParam ); } return cObjType\_Basic;

* Support for ECM\_GETCOMPICON must be restructured to return the HBITMAP for the control. The code for this, may be:

###### // wParam is true if the library is to be loaded. This enables fastest // load time as it avoids loading the bitmap for every library that // the handler supports. if ( wParam ) {   CONTROL\* control = getControl( pEci->mCompLibId, pEci->mCompId );   if (control)   {

###### EXTfldval exfldval;     exfldval.setLong( (qlong) control ->getHBitMap() );     ECOaddParam(eci,&exfldval);     // Bitmap returned     return qtrue;   } }

###### // No bitmap returned return qfalse;

* Support for ECM\_GETCONSTNAME must be restructured. Obviously control constants are not in the handlers' resources, so the constant list returned to Omnis must be manually built. An example of this would be:

###### CONTROLLIB\* curLib = getLib( pEci->mCompLibId ); if ( curLib ) {   EXTfldval extfldval;   EXTqlist list; list.clear( listVlen );   for ( qshort constCount=1; constCount<=curLib->mConstantCount; constCount++ )   {     EXTfldval cval; qlong line = list.insline();     // Constant ID     list.getColValRef( line , 1, cval, qtrue );     cval.setLong( curLib->getConstId(constCount) );     // Constant String     list.getColValRef( line , 2, cval, qtrue );     cval.setChar( curLib->getConstantName(constCount) );   }   extfldval.setList( list, qtrue);   ECOaddParam(pEci,&extfldval);   return qtrue; } // No constants return qfalse;

* Support for ECM\_GETPROPNAME, ECM\_GETEVENTNAME, ECM\_GETMETHODNAME must be restructured to return the properties for a control. An example of this would be:

###### CONTROL\* cntrl = getControl( pEci->mCompLibId, pEci->mCompId); if (cntrl) {   EXTfldval extfldval;   EXTqlist list; list.clear( listVlen );   for ( qshort num=1; num <= cntrl->getCount(); num ++ )   {     EXTfldval cval;     qlong line = list.insertLine();

###### // External id     list.getColValRef( line , 1, cval, qtrue );     cval.setLong(cntrl->getId(num));

###### // Name     list.getColValRef( line, 2, cval, qtrue );     cval.setChar(cntrl ->getName(num) );

###### // fft type of property/return type     list.getColValRef( line , 3, cval, qtrue );     cval.setLong(cntrl->getType(num) );

###### // EXTD\_ flags     list.getColValRef( line, 4, cval, qtrue );     cval.setLong(cntrl ->getFlags(num) );

###### if ( ECM\_GETPROPNAME==message )     {       // For properties you need to set the constant range       // Const Start (zero if none)       list.getColValRef( line , 6, cval, qtrue );       cval.setLong(cntrl->getConstStart(num));

###### // Const End (zero if none)       list.getColValRef( line , 7, cval, qtrue );       cval.setLong(cntrl->getConstEnd(num));

###### }     else

###### {       // For functions & events you need to add a       // list containing parameters       EXTqlist paramlist;       paramlist.clear( listVlen );       for ( qshort m=1; m<=cntrl->getParamCount(); m++ )       {         qlong paramline = paramlist.insertLine();         // Parameter name         paramlist.getColValRef( paramline, 1, cval, qtrue);         cval.setChar(cntrl->getParamName(m));

###### // fft Data type         paramlist.getColValRef( paramline, 2, cval, qtrue);         cval.setLong(cntrl->getParamType(m));

###### // EXTD\_ flags         paramlist.getColValRef( paramline, 3, cval, qtrue);         cval.setLong( cntrl->getParamFlags(m) );       }       list.getColValRef( line , 6, cval, qtrue );       cval.setlist( paramlist, qtrue );     }   }   extfldval.setList( list, qtrue);   ECOaddParam(pEci,&extfldval);   return qtrue; } // No properties return qfalse;

* Finally, on receipt of the ECM\_OBJCONSTRUCT, the control handler needs to construct the appropriate control. To enable this, the members mCompId and mCompLibId in the EXTCompInfo structure will contain the unique identifiers as declared during ECM\_GETCOMPLIBINFO and ECM\_GETCOMPID messages.

See also ECM\_CONNECT, ECM\_GETHANDLERICON, ECM\_GETCOMPLIBINFO, ECM\_GETCOMPID, ECM\_GETCOMPICON, ECM\_GETCONSTNAME, ECM\_GETPROPNAME, ECM\_GETEVENTNAME, ECM\_GETMETHODNAME.

## Background Components

When creating a background external component, you need to be aware of the differences between real components, and of the extra messages you may need to respond to.

A background component is created in a different way within Omnis during runtime and design mode. When you are designing a background component in design mode, the component will be given a child window ( HWND ) to draw within. During design mode, Omnis maintains this child window. During runtime, no child window is created. Omnis will call your object to paint, in an existing window at a certain location. Given this runtime/design mode difference, you should not use any HWND API that requires a window, such as WNDsetCapture() as you may not have a valid child window.

The following messages describe the differences or meaning when received by a background component.

### ECM\_OBJCONSTRUCT

ECM\_OBJCONSTRUCT is sent to all component types. For background components you can test the wParam parameter and the ECM\_WFLAG\_NOHWND flag to tell if you are being created during design or runtime. For example :

###### case ECM\_OBJCONSTRUCT: {   tqfTile\* object = new tqfTile( hwnd );   object->mIsRealHWND = !(wParam & ECM\_WFLAG\_NOHWND);   ECOinsertObject( eci, hwnd, (void\*)object, wParam );   return qtrue; }

The above example creates a new background component object, and stores a flag in the class so the control knows if it has a real child window or not.

### ECM\_CONNECT

The ECM\_CONNECT needs to be handled for background external components. When Omnis calls your component with this message, the following code should be used. If the code is omitted, Omnis will create the control as a first class foreground object.

###### case ECM\_CONNECT: {   return EXT\_FLAG\_LOADED | EXT\_FLAG\_BCOMPONENTS; }

### ECM\_PRINT

ECM\_PRINT is a very important message. Normally with standard components you pick up the WM\_PAINT message so you can paint your control. During runtime, as you do not have a child window, you will never receive a WM\_PAINT message. During design mode you do have a child window, so in theory you could get a WM\_PAINT message, but you will not. To help background components keep a simple interface, Omnis sends only ECM\_PRINTto your component during runtime and design mode to indicate that it needs to be painted. A WNDpaintStructis passed in the **lParam** parameter which holds the area that needs painting and a HDC to paint within.

###### case ECM\_PRINT: {   tqfTile\* object = (tqfTile\*)ECOfindObject( eci->mOmnisInstance, hwnd, wParam );   WNDpaintStruct\* ps = (WNDpaintStruct\*)lParam;   if ( object ) object->paint( ps->hdc, &ps->rcPaint );     return qtrue;  }

The above example shows how to paint you background object in runtime or design mode.

## Web Client Components

Writing Web Client components is almost identical to writing standard window components. In fact, you can build both from the same source. There are however some small differences.

* You will need to link against a different set of libraries. Use the example project files as a guide.
* The final DLL/shared library name must match the component library name as specified by your resources. The library name resource ID is returned by ECOreturnCompInfo as a response to the ECM\_GETCOMPLIBINFO. As a rule of thumb, all our web client component names start with “FORM”, i.e. FORMTREE, FORMTIME, etc.
* Web client components must respond to the ECM\_GETCOMPSTOREGROUP message and return the group name with ECOreturnCStoreGrpName ECM\_GETVERSION (see Chapter 2—Components Reference). The group name must be “WEB Components” for web controls and “WEB Background Objects” for web background objects.
* Web client components must be data bound in order to manipulate Omnis data. There is no other way of telling the client that data has changed and needs to be sent to the server for the next event. There is a message ECM\_HASPRIMARYDATACHANGED (see Chapter 2—Components Reference) which the component needs to implement. You use it to tell the web client if the primary data has been changed by the user. If the component only displays data, it can have non-data bound properties which take instance variable names, but the component will not know when the data has changed.
* Web client components need to implement the following additional messages which deal with focusing and mouse clicks.   
  ECM\_CANFOCUS  
  ECM\_ CANCLICK  
   (see Chapter 2—Components Reference)
* In addition, web client components must implement a proper versioning system. There is a new ECOreturnVersion function which must be used as a response to ECM\_GETVERSION (see Chapter 2—Components Reference).
* ECOsendEvent function will always return true when called from web client components. In order to receive a result, the component must implement the ECM\_EVENTRESULT message (see Chapter 2—Components Reference).
* Some functions or classes require additional parameters when used from web client components. These are  
  EXTBMPref::EXTBMPref  
  EXTCURref::EXTCURref  
  (see Chapter 3—EXTBMPref/EXTCURref Class Reference)
* The EXTfile class and related functions are currently not supported.

The new generic stationary in the MACIDE folder (Mac only) includes basic code needed for writing web client components.