Additional information about ObjectWindows 2.5

This file contains a list of the new features for Borland's ObjectWindows for C++ version 2.5 and information about the product that is not included in the documentation. The online Help contains updated information not found in the printed documentation. In addition, the online Help contains information that was previously noted in this OWLDOC.WRI text file. For the latest information about Borland C++ and for information about how to contact Borland, see the README.TXT file included with your distribution disks.

This is a Windows Write document; the formatting information will be lost in viewers or editors that do not support Windows Write format. If you use Write to view this file, you can click the Maximize button in the upper right corner to enlarge the window. To move through the file, click the arrows at the top and bottom of the scroll bar on the right side of the file or use the PageUp or PageDown keys. To print this file, choose the Print command from the File menu.

TABLE OF CONTENTS

1. New features

ObjectWindows 2.5 includes the following new features:

- Complete encapsulation of OLE 2 using ObjectComponents
- Templates for ObjectComponents classes
- New classes that manage UI handles and hatched brushes
- 32-bit programs can use existing VBX controls
- New data type definitions
- New 32-bit dispatch functions

2. ObjectWindows classes

This section describes changes that have been made to ObjectWindows. Read the printed documentation, the DynaText viewer (available on the CD-ROM version of Borland C++), and the online Help for updated information about these changes.
LPSTR and LPCSTR data types have been changed to char far* and const char far* respectively.
TSortedStringArray has changed from a typedef to a class.
The class TMenuDescriptor is now derived from TMenu. Both TMenu and TMenuDescriptor now support deep copies.

The online Help includes the most updated information about the ObjectWindows API.
See the online Help for a description of the following classes, functions, enums, and constants not included in the printed documentation:
ObjectWindows classes:
TRegLink, TRegList, TRegItem, TOleLinkView
ObjectWindows functions:
TWindow::RouteCommandEnable
TOleWindow::EvPaint
TOleWindow::EvOcViewPartActivate
TOleWindow::EvOcViewBreakLink, TOleWindow::EvOcViewSetLink
TOleView::EvOcViewBreakLink, TOleView::EvOcViewSetLink
TWindowFlag has a new enumerated value: wfDeleteOnClose

TVbxEventHandler::EvVbxInitForm
ObjectComponents classes:
TAutoDoubleRef, TAutoFloatRef, TAutoInt, TAutoLongRef, TAutoShortRef, TOcDataProvider, TOcFormat, TOcFormatInfo, TOcPartChangeInfo, TOcStorage, TOcStream, TOcChangeInfo, TOcFormat, TOcLinkView, TOcViewCollection, and TOcViewCollectionIter
ObjectComponents data types:
FILETIME, STGM_xxxx constants, STGC enum, STATSTG struct, appname Registration Key, AUTOPROXY macro, Ocxxxx global function
OC_VIEWxxxx Messages includes 4 new constants: OC_VIEWBREAKLINK, OC_VIEWPARTACTIVATE, OC_VIEWSETLINK, OC_VIEWSETTITLE
AutoDataType enum includes 2 new bit flags: atSafeArray and atAutoClass
The online Help and the DynaText viewer (available on the CD-ROM version of Borland C++) list the correct See Also references for the operator member functions in the TRect class.
The following function, which is described in the printed documentation, has been removed:
TOleWindow::Paint

Avoid using TDC::SaveDC and TDC::RestoreDC when GDI objects have been selected.

3. ObjectComponents Framework (OCF) classes

ObjectWindows includes a new set of classes for making programs that support the following OLE capabilities:

- Linking and embedding containers
- Linking and embedding servers
- Automation servers (automatable servers)
- Automation controllers
- OLE clipboard operations
- OLE drag and drop operations
- In-place editing
- OLE user interface, including menu merging, pop-up menu for activated object, verbs on container's Edit menu
- Compound file storage
- Registration
- Localized strings for international support
- Type libraries

For additional information about OjectComponents, see the section on ObjectComponents Framework under "Important Information" in the README.TXT file included with your distribution disks.

4. ObjectWindows 1.0 compatibility

If your OWL 1.0x applications used TNSCollection (defined in tcollect.h) classes, you will need to convert your code so that it can use the new BIDS classes, for example TArrayAsVector, TListImp, and so on.

The sample file, ..\OWL\OWL_1\OLDFILEW, is designed to be compatible with OWL1.0 applications and shouldn't be used for 32-bit applications. Users who have OWL 1.0 applications using the obsolete TFileWindow class can build and use this library to support their applications.

5. Handling OWLCVT header file names and DefaultProcessing

When it converts your files, OWLCVT adds a relative path name to the #include directive for any OWL 2.5 header files that are found in your files. For example,

#include <button.h> becomes #include <owl\button.h>

This change was made so that your converted application is able to use the simplified include paths that are used throughout BC4.5. If your code uses any header file names that are the same as any of the header file names for the Borland C++ version 3 class library or ObjectWindows library, this change will cause your code to compile incorrectly.

To remedy this situation, remove the relative path name that OWLCVT added from the #include directive in each case where there are conflicting header names. You should also rename these files to avoid future conflicts. (Of course, any headers that you #included in your OWL 1.0x application that no longer exist in OWL 2.5 will need to have their #include directives removed for a successful compilation.)

If you are overriding any of these functions - DefCommandProc, DefChildProc, DefWndProc, and DefNotificationProc - in derived classes, OWLCVT turns them into DefaultProcessing. This causes no problems if you're just calling the base class versions of these functions. However, if you override these functions in your derived classes, they are also changed to DefaultProcessing. Therefore, you will need to change these function names before running OWLCVT (for example, change DefCommandProc to MyCommandProc) so OWLCVT doesn't change them.

Appendix A in the ObjectWindows *Programmer's Guide* specifies that your header file paths for conversion should be in the following order and consist of the following paths:

```
-Ic:\bc45\include\owlcvt
-Ic:\bc45\include\classlib\obsolete
-Ic:\bc45\include
```

In most cases, this will suffice. However, depending on whether or not you make use of certain version 3 class library classes in your OWL 1.0x application, you may also need to point to the new class library include directory, in the following order:

```
-Ic:\bc45\include\owlcvt
-Ic:\bc45\include\classlib\obsolete
-Ic:\bc45\include\classlib
-Ic:\bc45\include
```

This additional path is necessary if your OWL 1.0x code references class library header files that existed in the version 3 libraries and have now been moved to the new (version 4.5) location. Once OWLCVT has completed its conversion process, the first three of these paths should no longer be necessary, that is, -Ic:\bc45\include should suffice.

6. ObjectWindows streaming

It's important to distinguish between two types of objects -- resident objects which may be streamed out but which are not reconstructed when streamed back in, and dynamic objects which are reconstructed when streamed in. Resident objects include static objects and those objects which are present when the application starts, such as its main window. These objects must be streamed out by reference rather than via a pointer. Dynamic objects, on the other hand, must be streamed out via pointers, causing them to be reconstructed when streamed back in. It is essential that resident objects be streamed out before any objects which stream out pointers to the resident objects, else duplicate objects will be constructed when streaming in.

When streaming objects in, the Streamer::Read function must insure that all data fields are initialized, as the streaming constructor does not set any of the data fields. The member data that does not get streamed in must be set to meaningful values. Care must be taken to initialize the members before streaming in base class data or pointers to objects which have pointers back to the current object. As opposed to constructor code, virtual functions are enabled in Streamer::Read.

In ObjectWindows, window objects stream their parents, siblings, and children. To prevent the streaming code from walking up though resident windows, flags are checked on certain windows to determine to top of the window tree. For SDI applications, the main window (marked by the

wfMainWindow flag) is by default resident and not streamed; for MDI applications, the MDI client window (marked by the wfStreamTop flag) is by default resident and not streamed. Even though no data is actually streamed from these top windows, they must be streamed by reference before any other windows are streamed which might contain pointers to them. To cause all the child windows to be streamed, stream out any child window by pointer, using a call such as GetFirstChild() or GetActiveMDIChild(). For Doc/View applications the document manager must be streamed first, which will eventually stream all the windows associated with the views.

TModule objects must be streamed out before any other objects that contain pointers to the TModule objects. This includes the application (derived from TModule), which is generally streamed first. This insures that dynamic libraries are loaded before references are made into them. Note that streamed module pointers to user DLL's must use the application's TModule alias, NOT the TModule alias created within the DLL. When streaming ObjectWindows applications built with the DLL model, a reference to the ObjectWindows library will be automatically streamed by the TApplication streaming code.

The general order for streaming an ObjectWindows desktop state is:
1. From a command, stream the derived TApplication object by reference,
   e.g. os << *this;
2. The application object Streamer::Read/Write should call the base class,
   e.g. WriteBaseObject((TApplication*)GetObject(), os);
3. The application object streamer should then stream other relevant values,
   such as the main window size and location.
4. The application object streamer should then stream the client window by
   reference, e.g. os << *GetObject()->Client;
5. A Doc/View application object stream should then stream the document
   manager by reference, e.g. os << *GetObject()->DocManager;
6. A non-Doc/View application object streamer should stream the child windows
   by pointer, e.g. os << GetObject()->Client->GetActiveMDIChild();

See the MDISTRM and DOCVIEW examples under \EXAMPLES\OWL\OWLAPI directory for an illustration of these procedures.

To reduce the memory requirements of statically-linked ObjectWindows programs, the streaming code for most classes is linked in conditionally. As a result, unless a class streamer is explictly reference using a streaming operator or from another streamer, the steamer must be forced into the link using the _OWLLINK() macro, specifying the class name prefixed with an "r", e.g. _OWLLINK(rTMDIChild); Few, if any, of these macros are required, since class streamers generally reference their base classes and stream out their member data. Streaming the document manager will reference TDocument, and TView, but not the derived classes, TFileDocument, TEditView, and TListView.

7. Writing exception-safe code

When you write exception-handling code, keep in mind the following issues. When exceptions are thrown, the runtime stack unwinds, skipping over the unexecuted code of the intervening functions, until a matching catch block is encountered. Unless exceptions have been disabled, destructors are called for any automatic objects, that is, those constructed on the stack. However, objects created using "new" are not automatically destructed unless a "delete" call is explictly added to the destructor of any automatic object within the function. Note that if an exception is thrown during construction, destructors will be called for the base classes but not for a partially constructed subclass.

In ObjectWindows, an application's return status is propagated as follows:
- The message loop exits from either a WM_QUIT message or an unhandled exception handler with an abort status. The status code is returned.
- TApplication::Run() returns the status code from the message loop, or, if an exception is caught,

from the unhandled-exception handler.
- In your own OwlMain(), you can examine the return status and catch unhandled exceptions.

See XCPTVIEW.CPP, which implements class TExceptionView, for an example of how to use exception-handling functions. This file is in the ..\OWLAPI\DOCVIEW directory. For a detailed description of exception-handling functions, see the Borland C++ Library Reference.

8. Writing multi-threaded applications

OWL 2.5 provides some simple hooks for use when writing multi-threaded applications. These hooks are not intended to be a complete solution to the programming problems introduced by multi-threading, but they do provide a mechanism that allows a programmer who understands multi-threading to write robust multi-threaded OWL applications.

The multi-threading support in OWL follows the model of a data logging application: OWL itself uses only one thread, and that thread should always be the main thread of the application. Additional threads must synchronize with the OWL thread whenever they need to access any data that is accessible to the OWL thread. They can, of course, perform computations asynchronously from OWL as long as those computations do not rely on or change any of OWL's data.

In practice, this usually means writing a class whose data can be accessed both by OWL and by the additional thread. As long as the thread does not use data or functions outside of this class the problems of synchronization are well localized and can be controlled fairly easily. See the MTHREAD sample program for examples of how this is done.

OWL's Thread Control Mechanism

Each OWL application has a mutex semaphore that OWL locks whenever OWL is active. Additional threads should use this semaphore to gain access to OWL when needed. This is most easily done through the class TAppLock, which is a member of TApplication. TAppLock's constructor takes a reference to a TApplication object and requests a lock on that object's mutex semaphore. If the semaphore is locked by another thread when the request is made the constructor blocks until the semaphore is available. TAppLock's destructor releases the lock. A typical program would do something like this:

```
class DataWindow : public TWindow
{
public:
        void LogData();
};

void DataWindow::LogData()
{
        for(;;)
                {
                // start out with processing that does not involve OWL
                Sleep(200);

                // wait for the OWL semaphore
                TApplication::TAppLock( *GetApplication() );

                // finally, do processing that requires access to OWL
                Invalidate();
                }       // the semaphore is released by the destructor
}
```

Direct Access to the Application's Mutex Semaphore

In some cases it is necessary to access the application's mutex semaphore directly. For example, if the user tries to shut down the application while another thread is running, the application has to shut down the other thread. Under WIN32 this can be done by calling TerminateThread(), but that's rather drastic. It's usually better to signal the thread that the application is terminating, then wait for the thread to terminate itself. That lets the thread do any necessary cleanup before ending. This is the mechanism that the MTHREAD example uses: the main loop in each thread waits for two semaphores. One is an event semaphore, and when that semaphore is triggered by the application it indicates that the thread should terminate. The other is the OWL mutex semaphore. When that semaphore becomes available it means that OWL is not executing, and it is safe for the thread to call OWL functions. In order to do this, the HANDLE for the OWL mutex semaphore must be accessible. TApplicaton::GetMutex() provides a reference to the TMutex object that contains the mutex semaphore. The HANDLE to the semaphore can be obtained from the TMutex object.

9. Known problems

Using modal dialogs in OLE2 servers
~~~~~~~~~~~~~~~~~~~~~~~~~~~~
When an ObjectWindows application needs to open a modal dialog at the application level, it's typical to parent the dialog to the MainWindow. This technique will not work for OLE 2 servers when the application is servicing an in-place edit. The dialog needs to be parented to a window in the container's window hierarchy. One example of a window that serves as a parent is the server's view window itself; another one is the container's frame; a third choice is the window with the focus. If all of the server command handlers that use modal dialogs are implemented in the server view class, it is best to use the view window itself as the parent. If the view window is not available (such as for commands handled by the application), the following technique can be used:

```
 TDialog(&TWindow(::GetFocus()this), IDD_ABOUT)).Execute();
```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~