ObjectStore® PSE Pro™

Building Applications With PSE Pro for C++

Release 6.3

PROGRESS SOFTWARE | Real Time Division

Building Applications with PSE Pro for C++, Release 6.3, October 2005

September 2005

Contents

Preface

Purpose

Building Applications with PSE Pro for C++ provides information and instructions for building database applications that use PSE Pro.

Audience

This book is for programmers responsible for writing database applications that use ObjectStore PSE Pro for C++. No previous knowledge of PSE Pro for C++ or the C++ interface to ObjectStore is required. It is assumed that you are experienced with Visual C++.

Scope

This book supports Release 6.3 of ObjectStore PSE Pro for C++.

How This Book Is Organized

Chapter 1, Overview of Building PSE Pro Applications, on page 9, lists the components each PSE Pro application must include and briefly describes the steps for building an application.

Chapter 2, Generating the Application Schema, on page 11, provides instructions for creating the schema source file and running the schema generator.

Chapter 3, Compiling Your Source Files, on page 17, describes the compiler switches for compiling your PSE Pro application.

Chapter 4, Linking Your Application, on page 19, provides the names of the libraries you must link with and discusses some schema generation problems that could potentially cause link errors.

Chapter 5, PSE Pro Static Builds, on page 23, describes how to link with the static version of PSE Pro.

Chapter 6, Using MFC and MSDEV IDE, on page 25, discusses how to use PSE Pro in the MSDEV IDE.

Notation Conventions

This document uses the following conventions:

Convention	*Meaning*
`Courier`	`Courier font` indicates code, syntax, file names, API names, system output, and the like.
`Bold Courier`	**`Bold Courier font`** is used to emphasize particular code, such as user input.
`Italic Courier`	*`Italic Courier font`* indicates the name of an argument or variable for which you must supply a value.
Sans serif	Sans serif typeface indicates the names of user interface elements such as dialog boxes, buttons, and fields.
Italic serif	In text, *italic serif typeface* indicates the first use of an important term.
`[ ]`	Brackets enclose optional arguments.
`{ }*` or `{ }+`	When braces are followed by an asterisk (`*`), the items enclosed by the braces can be repeated `0` or more times; if followed by a plus sign (`+`), one or more times.
`{ a | b | c }`	Braces enclose two or more items. You can specify only one of the enclosed items. Vertical bars represent OR separators. For example, you can specify *a* or *b* or *c*.
`...`	Three consecutive periods can indicate either that material not relevant to the example has been omitted or that the previous item can be repeated.

Progress Software Real Time Division on the World Wide Web

The Progress Software Real Time Division Web site (`www.progress.com/realtime`) provides a variety of useful information about products, news and events, special programs, support, and training opportunities.

Technical Support

To obtain information about purchasing technical support, contact your local sales office listed at `www.progress.com/realtime/techsupport/contact`, or in North America call 1-781-280-4833. When you purchase technical support, the following services are available to you:

- You can send questions to `realtime-support@progress.com`. Remember to include your serial number in the subject of the electronic mail message.
- You can call the Technical Support organization to get help resolving problems. If you are in North America, call 1-781-280-4005. If you are outside North America, refer to the Technical Support Web site at `www.progress.com/realtime/techsupport/contact`.
- You can file a report or question with Technical Support by going to `www.progress.com/realtime/techsupport/techsupport_direct`.
- You can access the Technical Support Web site, which includes

- A template for submitting a support request. This helps you provide the necessary details, which speeds response time.
- Solution Knowledge Base that you can browse and query.
- Online documentation for all products.
- White papers and short articles about using Real Time Division products.
- Sample code and examples.
- The latest versions of products, service packs, and publicly available patches that you can download.
- Access to a support matrix that lists platform configurations supported by this release.
- Support policies.
- Local phone numbers and hours when support personnel can be reached.

Education Services

To learn about standard course offerings and custom workshops, use the Real Time Division education services site (`www.progress.com/realtime/services`).

If you are in North America, you can call 1-800-477-6473 x4452 to register for classes. If you are outside North America, refer to the Technical Support Web site. For information on current course offerings or pricing, send e-mail to `classes@progress.com`.

Searchable Documents

In addition to the online documentation that is included with your software distribution, the full set of product documentation is available on the Technical Support Web site at `www.progress.com/realtime/techsupport/documentation`. The site provides documentation for the most recent release and the previous supported release. Service Pack README files are also included to provide historical context for specific issues. Be sure to check this site for new information or documentation clarifications posted between releases.

Your Comments

Real Time Division product development welcomes your comments about its documentation. Send any product feedback to `realtime-support@progress.com`. To expedite your documentation feedback, begin the subject with `Doc:`. For example:

```
Subject: Doc: Incorrect message on page 76 of reference manual
```

Chapter 1
Overview of Building PSE Pro Applications

This chapter lists the requirements for each PSE Pro application and the steps you must follow to build a PSE Pro application or library.

Requirements for Each PSE Pro Application

The following elements are required in every PSE Pro application:

- Including the PSE Pro Header File on page 9
- Establishing Fault Handlers on page 9
- Initializing the PSE Pro Run Time on page 10

Including the PSE Pro Header File

For a program to use any PSE Pro features, it must include the file `os_pse/ostore.hh`.

Establishing Fault Handlers

PSE Pro must handle all memory access violations, because some are actually references to persistent memory in a database. On Windows, there is no function for registering a signal handler for such access violations that also works when you are debugging a PSE Pro application (the `SetUnhandledExceptionFilter()` function does not work when you use the Visual C++ debugger).

Every PSE Pro application must put a handler for access violation at the top of every stack in the program. This normally means putting a handler in a program's `main()` or `WinMain()` function and, if the program uses multiple threads, putting a handler in the first function of each new thread.

PSE Pro provides two macros to use in establishing a fault handler:

- `OS_PSE_ESTABLISH_FAULT_HANDLER`
- `OS_PSE_END_FAULT_HANDLER`

On UNIX platforms PSE Pro can register a signal handler for such access violations. The handler needs to be registered only once using the macros `OS_PSE_ESTABLISH_`

FAULT_HANDLER and OS_PSE_END_FAULT_HANDLER. You can also invoke the macro for every thread, as on Windows, with no ill effects.

Initializing the PSE Pro Run Time

To use the PSE Pro API, you must first call the static member function objectstore::initialize(). The function signature is

```
static void initialize() ;
```

A process can execute initialize() more than once. After the first execution, calling this function has no effect.

Other Requirements

Transactions are optional in PSE Pro. If your application will use the PSE Pro transaction facility, you must call the static member function os_transaction::initialize().

If your application will use the PSE Pro collections facility, you must call the static member function os_collection::initialize().

If you are using the Command Prompt to build with NMAKE, ospsevars.bat has to be executed in order to update the PATH, INCLUDE, and LIB environment variables.

The installation program modifies various *Directory* options in the Visual Studio IDE (view modifications at Tools | Options, Directories tab), so that the examples can be built in the IDE.

During installation for Windows XP and Windows 2000, the ospsevars.bat file and the PSE Pro DLLs are copied into the PSE Pro installation's bin directory. By default the installation directory is c:\odi\PSEPROC6.3.

The PSE Pro schema generator tool, pssg, calls the VC++ compiler cl.exe. Make sure that the Executable files option in the Visual Studio IDE (Tools | Options, select the Directories tab) contains the correct bin directories.

Steps for Building a PSE Pro Application

Follow these steps to build a PSE Pro application:

1. Create a schema source file. See Creating Schema Source Files on page 11.
2. Run the PSE Pro schema generator, pssg, to generate a schema object file. See Running the Schema Generator on page 13.
3. Run your C++ compiler on your application or library source files. See Compiling Your Source Files on page 17.
4. Link your application or library with the schema object file and the PSE Pro library. See Linking Your Application on page 19.

Chapter 2
Generating the Application Schema

To build your application, you must link your application object files with a schema object file that contains information about the persistent classes in your application. This chapter provides information about how to create the schema source file and run the PSE Pro schema generator, `pssg`, on it to create the schema object file. It discusses the following topics:

Because the PSE Pro schema generator tool, `pssg`, does not run on UNIX platforms, the process for generating an application's schema differs depending on whether the application will run on Windows or UNIX platforms. For more details on how to use `pssg` to generate application schema files on the different UNIX platforms, see the `readme` files in the installation directory of the specific UNIX products.

Creating Schema Source Files

An application's *schema* consists of the classes of objects in the databases it uses. To create the schema, every PSE Pro application or library must be linked with an object file that contains information about the classes of persistent objects used by the application. You generate this object file from the *schema source file*, which is a simple source file that you create as follows:

- Include the PSE Pro header file `<os_pse/ostore.hh>`.
- Include the definition of each class of object the application might store in a database, as well as the definition of each class of object in each database the application might open.
- Call the `OS_MARK_SCHEMA_TYPE()` macro for each included class and supply the class name as argument. Do *not* put quotation marks around the class name. Enter the argument *without white space*. The macro cannot span more than *one (1)* line in the schema source file.

- By convention, the schema source files distributed with the PSE Pro example applications use the `.scm` extension.

Here is an example:

```
/*** schema.scm ***/
#include <os_pse/ostore.hh>
#include "image_map.h"
#include "image.h"
#include "document.h"
OS_MARK_SCHEMA_TYPE(image_map);
OS_MARK_SCHEMA_TYPE(image);
OS_MARK_SCHEMA_TYPE(document);
```

If a nested class is public, calling `OS_MARK_SCHEMA_TYPE()` for the outer class is sufficient. The schema generator automatically generates schema for the nested class. When you define a class inside a `private` or `protected` section of another class's definition, the class is not visible to the schema generator, even if you mark the containing class. For classes embedded in private or protected sections, call `OS_MARK_SCHEMA_NESTED_TYPE()` instead of `OS_MARK_SCHEMA_TYPE()`.

You must mark each instantiation of any class template you use. In the following example, `BinaryTree` is a class template, so you must mark `BinaryTree<Data>`. Here is a schema source file that marks `BinaryTree<Data>` and `Data`:

```
/*** schema.scm ***/

#include <os_pse/ostore.hh> // PSE Pro header file
#include "BinaryTree.cpp"
#include "Data.h"

// Mark the classes to allow persistent allocation
OS_MARK_SCHEMA_TYPE(BinaryTree<Data>);
OS_MARK_SCHEMA_TYPE(Data);
```

Collections

For applications that will use the PSE Pro collections facility, include the PSE Pro header file `<os_pse/coll.hh>` in the schema source file after `<os_pse/ostore.hh>`.

For applications that will use persistent dictionaries, include the PSE header file `<os_pse/coll/dict_pt.hh>` in the schema source file. You must call the macro `OS_MARK_DICTIONARY()` in the schema source file for each key-type/element-type pair that you use. Here is an example:

```
/*** schema.scm ***/
#include <os_pse/ostore.hh>
#include <os_pse/coll.hh>
#include <os_pse/coll/dict_pt.hh>
#include "employee.hh"
OS_MARK_SCHEMA_TYPE(employee);
OS_MARK_DICTIONARY(char*, employee*);
```

Running the Schema Generator

Every PSE Pro application or library must be linked with an object file that contains information about the classes of persistent objects used by the application. *Schema generation* is the process of generating this object file from the schema source file. You run the PSE Pro utility `pssg` on the schema source file to generate the schema object file. This section discusses the following topics:

Format of the Schema Generator Command Line

The format of the `pssg` command line is a bit different depending on whether you plan to run your application on Windows or UNIX.

Windows

To generate the schema object file for an application that will run on Windows, use the following `pssg` command line format (on one line):

```
pssg [ CCFLAGS ]
   [ -asof filename ]
   [ -asdll dllname -asdb dbname ]
   schema-source-file
```

`CCFLAGS`	Indicates that you want `pssg` to pass the specified compiler switches to the C++ compiler when it compiles the schema source file during schema generation, Replace *`CCFLAGS`* with the compiler switches you want to pass.
`-asof` *`filename`*	The default name of the generated schema object file is `osschm.obj`. If you want the schema object file to have a different name, specify the `-asof` option and replace *`filename`* with the name you want for the generated schema object file. The filename extension is always `.obj`.
`-asdll` *`dllname`*	Indicates that you want to create a schema library. Replace *`dllname`* with the name you want the schema library to have. If you specify the `-asdll` option, you must also specify the `-asdb` option.
`-asdb` *`dbname`*	Indicates the application schema database in which you want the `pssg` utility to store Metaobject Protocol (MOP) information that it uses to generate the schema library. Replace *`dbname`* with the path you want the application schema database to have. If you specify the `-asdb` option, you must also specify the `-asdll` option.

schema-source-file	Replace *schema-source-file* with the name of the schema source file for which you want to generate a schema object file. By convention, the schema source files distributed with the PSE Pro example applications use the `.scm` extension.

UNIX

To generate the schema object file for an application that will run on UNIX, use the following `pssg` command line format (on one line):

```
pssg
   [  CCFLAGS ]
   [ -ascpp filename ]
   [ -aslib library-name ]
   schema-source-file
```

CCFLAGS	Indicates that you want `pssg` to pass the specified compiler switches to the C++ compiler when it compiles the schema source file during schema generation, Replace *CCFLAGS* with the compiler switches you want to pass.
`-ascpp` *filename*	The default name of the generated schema object file is `osschm.cpp`. If you want the schema object file to have a different name, specify the `-ascpp` option and replace *filename* with the name you want for the generated schema object file. The filename extension is always `.cpp`. You must compile and link the generated schema object file with your application.
`-aslib` *library-name*	Indicates that you want `pssg` to create a schema library. Replace *library-name* with the name you want the schema library to have.
schema-source-file	Replace *schema-source-file* with the name of the schema source file for which you want to generate a schema object file. By convention, the schema source files distributed with the PSE Pro example applications use the `.scm` extension.

Using Schema Libraries

You can use the `pssg` utility to generate a schema library. For applications that run on Windows, a schema library is a dll. For applications that run on UNIX, a schema library is a shared library. You can use a schema library in the following ways.

- For verifying a database. See *PSE Pro for C++ API User Guide*, Checking Databases with the ospseverifydb Utility.
- In place of linking with the traditional schema object file in a PSE Pro application. When your program loads the schema library, PSE Pro initializes the typespecs for classes that are marked in the schema library.

When Regenerating the Schema Is Required

You should regenerate the schema object file whenever you add or remove class definitions or modify a class definition in your application or library.

You can use a make rule like the following:

```
CC_OPTS = /DWIN32 /GX ...
schema-object-file: schema-source-file
  pssg $(CC_OPTS) -asof schema-object-file $@ schema-source-file
```

Compiler Switches Used by the Schema Generator

`pssg` uses the following compiler switches. In case of conflict with user-supplied flags, these override the user-supplied flags:

- `/MD`

  Use the `msvcrt.dll` run time.

- `/Od`

  Debug.

- `/Z7`

  MS C7 debug mode.

- `/Gx`

  Enable C++ exceptions.

- `/D_ODI_PSSG=1`

  Use this preprocessor macro to hide code from `pssg`. (ObjectStore's schema tool, `ossg`, uses `_ODI_OSSG_`).

Exporting vtbls

If you do not export a class with `declspec(dllexport)`, and the class is a persistent class that is marked in a schema source file, you must explicitly export the class's vtbl in a `.def` file. For example, the contents of the `.def` file must look like this:

```
EXPORTS
??_7Foo@@6B@
--and any other symbol you want explicitly exported--
```

Then specify the `/def` option to the linker and identify the `.def` file.

If you do not export the vtbl, you receive an unresolved symbol that is caused by the `osschm.obj` file created by the schema generator (`pssg`).

Schema Generation in the VC++ IDE

In the VC++ IDE, you can create a custom build rule to compile the *schema-source-file*:

1. In the FileView of the Workspace pane, right-click the schema source file (or select Project | Settings). The Project Settings dialog box is displayed.
2. Click the Custom Build tab (if the schema source file has a `.cpp` extension, you may need to click Always use custom build step on the General tab in order to display the Custom Build tab).
3. In the Commands text area, enter the following build command:

   ```
   pssg /asof $(INTDIR)\osschm.obj schema-source-file
   ```

4. In the Outputs text area, enter the following path name of the object file: `$(INTDIR)\osschm.obj`
5. In the Description text area, enter

   ```
   Generating ObjectStore/PSE typespecs
   ```

6. Select All configurations from the Settings For drop-down list so the build rule applies to both debug and release builds.

The PSE Pro `Quickstart` example shows how the custom build rule should look. To display this example, select Start | Programs | ObjectStore PSE Pro for C++ | QuickStart and follow steps 1 and 2, above.

Schema Generation on UNIX Platforms

The `pssg` schema generator for UNIX platforms does not directly generate schema object code; instead it generates a schema `cpp` file, which is then compiled and linked with the application. Here are the steps involved:

1. Run `pssg` against the schema source file with the `-ascpp` option.
2. Compile the generated schema source file with the C++ compiler to create the schema object file.
3. Link the application object files with the schema object file.

For more information about creating schemas for use by applications that run on UNIX platforms, see the readme file accompanying specific PSE Pro products.

Chapter 3
Compiling Your Source Files

This chapter includes information helpful in compiling source files on a variety of platforms.

Supported Platforms

PSE Pro is available in several versions. The table that follows lists the platforms supported by this release of PSE Pro for C++.

Platform	*Compiler*
HP-UX 11.11 (32-bit)	HP aC++ 3.56
HP-UX 11.11 (64-bit)	HP aC++ 3.56
Linux 2.4.18-14smp (32-bit)	GCC 3.2.3
Linux (64-bit)	GCC 3.2.3
Solaris (32-bit)	Sun One Studio 8
Windows XP and 2000	Visial C++ 7.1 (unmanaged)

Compilation Options

The following sections list the various options used when compiling PSE Pro applications on specific platforms. For UNIX versions of PSE Pro, the `readme` file in the PSE Pro installation directory contains additional information about compiling applications.

HP-UX 11.11

Use the HP C++ aCC A03.56 compiler with the following compiler switches:

- `+eh` to specify exception support, the default
- `-D_REENTRANT`
- `-g`, `- g0`, `-g1`, or `-O2` to specify `debug` or `opt`
- `+DAportable +DS2.0`
- `-I` to specify the PSE Pro include files
- `+Z` for PIC (not needed by PSE Pro)
- `-ext +DA2.0W` (64 bit)
- `+DA2.0N` (32 bit)

Linux

Use the GCC 3.2.3 compiler with the following compiler switches:

- `-g`, `-g0`, or `-g1` or `-O2` to specify `debug` or `opt`
- `-I` to specify the PSE Pro include files

Solaris (32 bit)

Use the Sun One Studio 8 compiler with the following compiler switches:

- `-mt` Compile for multithreaded code
- `-g/g0` or `-O2/-xO2` to specify debug or opt
- `-I` to specify the PSE Pro include files
- `-KPIC` for PIC (not needed by PSE Pro)

Windows

Use the Microsoft Visual C++ compiler with the following compile switches:

- `/Gx` to enable C++ exceptions
- `/MD` to use the `msvcrt.dll` run time (multithreaded DLL version), or `/MDd` to use the `msvcrtd.dll` `_DEBUG` run time

Chapter 4
Linking Your Application

This chapter discusses these topics:

Libraries You Must Link With

Windows

Use the Microsoft Visual C++ compiler to link PSE Pro applications and libraries. Link against the library `osclt.lib` and run against the DLL `o6psepr.dll`.

For debugging, link against the library `oscltd.lib` and run against the DLL `o6pseprd.dll`.

Upon inclusion, `os_pse/ostore.hh` sets the default library.

To use the MFC extension DLL, link with `osmfc.lib` or (for debug) `osmfcd.lib`, and run against `o6mfc.dll` or (for debug) `o6mfcd.dll`.

UNIX

When building applications with the UNIX versions of PSE Pro, you always need to link with the `lib/libospse` library (or its debug equivalent `lib/debug/libospse`). In addition, if your application uses collections you must also link with the PSE Pro collections library `lib/libospsecol` (or its debug equivalent `lib/debug/libospsecol`).

Use the `-L` option to specify the directory where the PSE Pro libraries are located and the `-l` option to specify individual libraries. When you specify a PSE Pro library, do not use the `lib` portion of the library name. For example, if the environment variable `OS_PSE_ROOTDIR` includes the location of the PSE Pro libraries, a command line might look like this:

```
CC -L$(OS_PSE_ROOTDIR)/lib -o my_app main.o schema.o -lospse
```

Linker Options

The following sections list the linker options to use when building PSE Pro applications on UNIX platforms. For more information on linker options for specific UNIX platforms, see the `readme` file in the installation directory.

HP-UX 11.11

- `+DAportable +DS2.0 Specify` portable architecture.
- `-L$(OS_PSE_ROOTDIR)/lib -lospse` Link with PSE Pro run time.
- `-g`
- `-Wl, -E`
- `-ext +DA2.0W` (64 bit)
- `+DA2.ON` (32 bit)

Linux

- `-L$(OS_PSE_ROOTDIR)/lib -lospse` Link with PSE Pro run time.
- `-g`
- `-rdynamic` Use global symbols in executable.

Solaris (32 bit)

- `-mt` Link for multithreaded code.
- `-g/g0` Prepare program for debugging.
- `-L$(OS_PSE_ROOTDIR)/lib -lospse` Link with PSE Pro run time.

Windows

On Windows platforms, no special linker options are required when linking PSE Pro applications.

Typical Link Errors Related to Schema Generation

If you get a link error like the following, your program uses the typespec (os_ts<Foo> in this case) for a class (Foo in this case), but your schema source file fails to mark the class.

```
osschm.obj : error LNK2001: unresolved external symbol  "private:
static class os_pse::os_ts<class Foo>
os_pse::os_ts<class Foo>::the_instance"
```

Include the class's definition and mark it in the schema source file.

If the above error occurs for os_ts<class Foo*> (indicating a pointer) rather than os_ts<class Foo>, you marked a pointer type. But pssg cannot create typespecs for a typed pointer. A workaround is to add the following os_ts<T> specialization (compare also to the stl42 example that defines STL list<Foo*>).

```
namespace os_pse {
  template <> class os_ts<Foo*> : public os_typespec {
  public:
    static inline os_typespec *get()
      { return os_typespec::get_pointer(); }
  };
  // define more pointer types:
  // template <> class os_ts<Bar*> : ...
  }
```

Yet another schema generation-related error that can occur during linking is an unresolved external symbol for a virtual function table. An example of this error is

```
osschm.obj : error LNK2001: unresolved external symbol
  "const Foo::`vftable'" (??_7Foo@@6B@)"
```

In this case, if class Foo has been marked in the schema source file, pssg creates a reference for a virtual function table if class Foo contains one or more virtual functions. If an application never calls the constructor of class Foo, the VC++ compiler does not create a vftable for that class. A way to force the creation of that symbol into the .obj file is to add the following invocation of the constructor to your program to ensure that the function _force_vftable() is never called:

```
void _force_vftable(void*) {
  _force_vftable( new Foo() );
  _force_vftable( ..more classes..);
```

Troubleshooting an Assertion Failure

The global `operator delete()` is defined in the PSE Pro DLL (`o5pse[pr][d].dll` and the `osclts*.lib` static libraries). Therefore, it depends on the linker to pick up this `operator delete()`.

For example, this failure appears in a pop-up dialog:

```
Microsoft Visual C++ Debug Library
Debug Assertion Failed!
Program: c:\temp\app.exe
File: dbgheap.c
Line: 1011
Expression: _CrtIsValidHeapPointer(pUserData)
For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts
(Press Retry to debug the application)
Abort   Retry   Ignore
```

When you run an application in the debugger, you might see this stack trace:

```
_CrtIsValidHeapPointer(void * 0x30001020) line 1612
_free_dbg_lk(void * 0x30001020, int 1) line 1011 + 9 bytes
_free_dbg(void * 0x30001020, int 1) line 970 + 13 bytes
free(void * 0x30001020) line 926 + 11 bytes
operator delete(void * 0x30001020) line 7 + 9 bytes
main() line 24 + 15 bytes
```

The argument to `operator delete()` is persistent: 0x30001020. To solve this problem, make sure that the linker reaches `osclt.lib` first, so that its `operator delete()` gets linked. A better solution is to define a `static inline operator delete` that is visible to all `.cpp` files that delete persistent objects. For example:

```
inline void operator delete(void* p)
{
  //calls free() on transient data
objectstore::free_memory(p);
}
```

This instructs your application to use PSE Pro's `operator delete()` function when the object is persistent.

Chapter 5
PSE Pro Static Builds

This chapter discusses how to build applications using the PSE Pro static libraries. Applications built this way can be executed without the PSE Pro run time libraries. The chapter contains the following topics:

Generating Schema for Static Builds

For static builds, make sure to add the `/D_OS_STATIC_LIB` compiler switch to `pssg`. If you do not, you see the following `LNK4049` warnings on `class os_basic_typespec` when linking statically with the generated schema object file. The warnings can be ignored.

```
LINK : warning LNK4049: locally defined symbol protected:

void __thiscall os_pse::os_basic_typespec::cleanup(void)
(?cleanup@os_basic_typespec@os_pse@@IAEXXZ) imported

void __thiscall os_pse::os_basic_typespec::initialize(void)
(?initialize@os_basic_typespec@os_pse@@IAEXXZ) imported
```

For all static builds, you see one other `LNK4049` warning when linking statically with the generated schema object file. This warning can also be ignored.

```
LINK : warning LNK4049: locally defined symbol

const os_pse::os_basic_typespec::vftable (
??_7os_basic_typespec@os_pse@@6B@) imported
```

Linking with the Static Version of PSE Pro

To link with the static version of the PSE Pro library, add the following compile switches:

- `/D_OS_STATIC_LIB=1`
- `/MD`, `/MDd`, `/MT`, or `/MTd`

Upon inclusion, `os_pse/ostore.hh` sets the default library based on these two compile switches. The PSE Pro installation contains the following static libraries built against different versions of the C run time:

- `osclts.lib`

  Use the multithreaded DLL version (`/MD`, `msvcrt.dll`).

- `oscltsd.lib`

  Use the debug, multithreaded, DLL version (`/MDd`, `msvcrtd.dll`).

- `oscltst.lib`

  Use the multithreaded static version (`/MT`, `libcmt.lib`).

- `oscltstd.lib`

  Use the debug, multithreaded, static version (`/MTd`, `libcmtd.lib`).

In the absence of `DllMain()` in the static PSE Pro libraries, the application must explicitly notify PSE Pro on thread creation and termination. If a thread does not access persistent data, the functions do not have to be called.

```
static void objectstore::thread_attach(HANDLE h)
static void objectstore::thread_detach(HANDLE h)
```

You can call `GetCurrentThread()` to obtain the handle `h`.

Chapter 6
Using MFC and MSDEV IDE

This chapter provides information about building your PSE Pro application when you use MFC and/or when you use MSDEV IDE. It discusses the following topics:

Using MFC

You can use MFC to build your PSE Pro application in one of two ways:

- If you are not going to allocate any MFC classes persistently, you can just use MFC as you would normally.
- If you want to use the MFC container classes persistently, use the persistent version of MFC, which consists of the MFC-style container and string classes in the PSE Pro MFC Extension DLL, `osmfc.dll`.

PSE Pro comes with an MFC AppWizard and an MFC extension DLL that has PSE Pro-specific subclasses of `CDocument`, `DOleDocument`, `COleLinkingDoc`, and `COleServerDoc`. The AppWizard generates applications whose document classes are derived from these subclasses.

The AppWizard also includes MFC-style container and string classes, for example, `CPSEPtrArray` and `CPSEString`. These classes have the same API as their MFC counterparts and also provide cast operators. See the `osmfc` project in `/src/osmfc` and its accompanying `docs/readme` file for details.

The `pse_scribble` example derives from `CPSEDocument` and uses a `CPSEPtrArray` to store `CStrokes`.

The PSE Pro document classes (`CPSEDocument`, `CPSEOleDocument`, `CPSEOleLinkingDoc`, and `CPSEServerDoc`) play the same role as their original MFC counterparts, except that they store the information in PSE Pro databases (either in regular files or in streams within OLE compound files).

This means that an MFC application that uses PSE Pro is essentially identical to a traditional MFC application, except that you do not have to implement any `Serialize()` methods. You can expect that opening a new document and saving a document will also be faster if you use PSE Pro because the entire document does not have to be loaded in and saved back out.

The complete project used to build the MFC extension DLL is available in the `src/osmfc` subdirectory under your installation directory. You can modify and redistribute this code as long as you maintain existing copyright notices.

The MFC-style container and string classes in the PSE Pro MFC Extension DLL (`CPSEPtrArray`, `CPSEString`, and others in `osmfc.dll`) supersede the previously released persistent MFC DLL (`mfcpse.dll`).

Using MSDEV IDE

The PSE Pro installation adds the following directories to the IDE's directory search paths and to your system environment:

- Include files: `c:\odi\PSEPROC63\include`
- Executable files: `c:\odi\PSEPROC63\bin`
- Library files: `c:\odi\PSEPROC63\lib`

Make sure that these directory files are set. In the Visual Studio IDE, select Tools | Options, and choose the Directories tab. Verify that the settings for Include files, Library files, and Executable files match the above directories.

Index

O

P

S

T

U

V