Cincom®

Cincom Smalltalk™ VisualWorks®

DLL and C Connect Developer's Guide

VisualWorks 8.3

P46-0112-13

SIMPLIFICATION THROUGH INNOVATION

Notice

Part Number: P46-0112-13

Software Release: 8.3

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About this Book

This Developer's Guide provides comprehensive instructions for using DLL and C Connect.®

The DLL and C Connect framework allows your Smalltalk application to invoke functions written using the C programming language, to create, modify, and use C language datatypes, and to send messages to Smalltalk objects from your C code. The C functions can either be statically linked into your application's executable, or dynamically loaded at run-time using the target platform's dynamic library loading facilities.

Audience

This Guide assumes you are familiar with the C programming language. All C code examples are written using ANSI C syntax.

Some chapters in this book also presuppose a general knowledge of object-oriented concepts, Smalltalk, the VisualWorks development environment, and its tools.

For an overview of Smalltalk, the VisualWorks environment and its application architecture, see the *Application Developer's Guide*.

Conventions

We follow a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Examples	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`c:\windows`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
filename.xwd	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks tools.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > **Open...**	Indicates the name of an item (**Open...**) on a menu (**File**).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	Select (or choose) a window location or a menu item, position the text cursor, or highlight text.

<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks window (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left Button	Left Button	Button
<Operate>	Right Button	Right Button	<Option>+<Select>
<Window>	Middle Button	<Ctrl> + <Select>	<Command>+<Select>

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and personal-use license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support.

Cincom provides all customers with help on product installation. For other issues, send email to: helpna@cincom.com.

Before contacting Technical Support, please be prepared to provide the following information:

- The release number, which is displayed in the Welcome Workspace when you start VisualWorks.
- Any modifications (patch files, auxiliary code, or examples) distributed by Cincom that you have loaded into the image. Choose **Help > About VisualWorks** in the VisualWorks Launcher

window. All installed patches can be found in the resulting dialog under **Patches** on the **System** tab.

- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, use **Copy Stack**, in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to Technical Support.
- The hardware platform, operating system, and other system information you are using.

You can contact Technical Support as follows:

E-mail	Send questions about VisualWorks to: helpna@cincom.com.
Web	Visit: http://supportweb.cincom.com and choose the link to Support.
Telephone	Within North America, call Cincom Technical Support at +1-800-727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time. Outside North America, contact the local authorized reseller of Cincom products.

Personal-Use Licensees

VisualWorks Personal Use License (PUL) is provided "as is," without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of users. To subscribe or unsubscribe, send a message to: vwnc-request@cs.uiuc.edu with the SUBJECT of **subscribe** or **unsubscribe**. You can then address emails to: vwnc@cs.uiuc.edu.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main window's menu bar (i.e., the Launcher window), and select one of the help options.

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks.

The guide you are reading is but one manual in the VisualWorks documentation library.

Complete documentation is available in the `/doc` directory of your VisualWorks installation.

Chapter

1

Tools and Techniques

Topics

This chapter gives an overview of DLL and C Connect, introducing you to the basic features of the product, and the special tools it provides for connecting your Smalltalk applications with C code modules. This chapter also presents a simple example to demonstrate the different techniques available for building interfaces to C modules.

Installing DLL and C Connect

Before you begin working with DLL and C Connect, make sure that you have first installed VisualWorks on your system. For more information on installing VisualWorks, see the *Application Developer's Guide*. Once VisualWorks has been installed, you may load the DLL and C Connect parcel to begin development.

External Interface Architecture

The purpose of DLL and C Connect is to enable your Smalltalk application to interact with code written using a C compiler. Your Smalltalk application can directly call C functions and directly modify C data objects. The DLL and C Connect interface provides the mechanism to allocate C data objects that adhere to a particular platform layout, and provides the mechanism to call C functions that use that platform's C calling conventions.

For multiprocessing applications, threaded function calls are fully supported. DLL and C Connect maps C data objects into Smalltalk objects, so not only can you use Smalltalk messages to manipulate C objects, but you can also share object references (pointers) between your C code and your Smalltalk application.

DLL and C Connect also enables your C code to send messages to Smalltalk objects. Finally, the product provides the mechanism to load (and link) dynamic-link libraries into your Smalltalk application's address space. This linking mechanism makes all the public entry points within the library available to your Smalltalk application.

The term function is used in this document as a generic term to represent code routines. Language specific terms include procedure, function, and subroutine.

DLL and C Connect introduces a new abstract class to the Smalltalk class hierarchy called `ExternalInterface`. You can define subclasses of the `ExternalInterface` class for the various C function and data object interfaces that your application must access. Your interface class is used both during the development of your application and during its deployment.

During development, the interface class provides a way to create the Smalltalk methods associated with the procedures, the data types, the macros, and the variables used by the external code. The interface class can be used to parse existing C header files and automatically generate all the interface methods. It enables you to load and unload libraries, and to package your interface into a parcel file so that it may be distributed to clients. Parcels are now the preferred mechanism for unloading and reloading interface classes; for a more detailed discussion, see the *Application Developer's Guide*. When your development is complete and your application is deployed, the interface class is used (typically transparently) by clients to access the external entry points. It is possible for your application to define and use multiple libraries, controlled with one or more interface classes.

The following diagram illustrates a Smalltalk application containing three interface classes. The first interface class encapsulates control of two C code modules. The second and third each encapsulate a single module. Module refers to C code statically linked to the Smalltalk application or contained in a dynamic-link library.

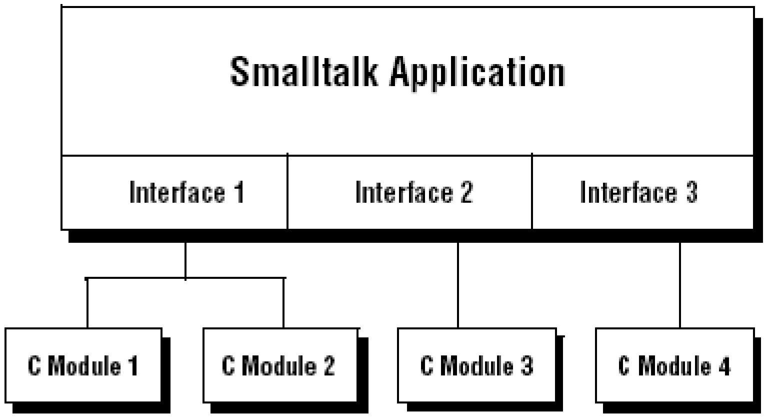


Figure 1: Smalltalk application model

Dynamic-Link Libraries

Vendors of C code modules typically package their products as dynamically-linkable libraries (DLLs) of functions. In this situation, DLL and C Connect provides the means to convert the state and behavior of Smalltalk objects to C data and functions in a DLL, and vice versa.

Dynamic-linking provides the following benefits:

- Smalltalk applications can take advantage of software provided by third-parties, such as commercial databases, network communications packages, or drivers for unusual devices.
- Smalltalk applications are not affected when library updates are distributed, as the application communicates with the library only through a well-defined interface.
- Smalltalk applications require less disk space as they do not need to contain the code that resides in the library.
- Multiple Smalltalk applications can share the same library. Because there will probably be only one copy of the library in memory, there will be fewer demands on physical memory and swap space.

Placing C Declarations into Smalltalk

DLL and C Connect extends the syntax of Smalltalk methods to place C language declarations directly into Smalltalk methods. However, this extended syntax is only available to subclasses of `ExternalInterface`. Examples of C language declarations that may appear as methods to subclasses of `ExternalInterface` appear below. The constructs are described here only as an introduction to the syntax extensions provided.

C #define statements

Use the `#define` methods to answer commonly used constant expressions. DLL and C Connect supports `#define` methods that evaluate to numbers or strings. The following example is a `#define` method that answers the `SmallInteger` 1024.

```
MAX_FILENAME_LENGTH
 <C: #define MAX_FILENAME_LENGTH 1024
 >
```

C type statements

Use `typedef` methods to answer type objects. These type objects are used to allocate C data objects, to build larger type structures, and to define C function prototypes. The following example answers an instance of the class `CTypedefType` that represents the `Point typedef` type declaration.

```
Point
 <C: typedef struct {
  float x;
  float y;
 } Point>
```

C function prototypes

Use function prototype methods to define a C function's argument types and return type. These prototypes are required so the Smalltalk execution machinery can correctly make a call to your C function. The following example declares a procedure, `addPoint()`, that accepts two `Point` type arguments and returns a `Point` type argument.

```
addPoint: arg1 with: arg2
 <C: Point addPoint(Point arg1, Point arg2)>
```

C variable declarations

Use variable declaration methods to define the global variable type. The first method shown below returns the value of the global variable and the second method sets the global variable's value to the given argument.

```
globalVariable
 <C: unsigned long globalVariable>

globalVariable: newValue
 <C: unsigned long globalVariable>
```

Accessing C Data Objects

DLL and C Connect also adds the ability to create and access C data objects. This occurs in two ways.

- By creating wrapper, or proxy, objects that enable your application to manipulate C data as if it were a Smalltalk object.
- By providing access to a new memory area called the external heap. C data objects that are allocated on the external heap are protected from movement and automatic reclamation by the Smalltalk memory manager.

A typical Smalltalk application is relieved of the standard memory management issues by the Smalltalk Object Engine. The Object Engine releases storage when it is no longer referenced, and compacts memory fragmented from repeated object allocation and deallocation. To allow a Smalltalk application to supply data pointers to C code, or to receive data pointers from C code that was not designed to work with an automatic memory manager, DLL and C Connect makes available several new strategies for managing application memory. In particular, it provides access to a new memory space that is under the control of the application rather than the Smalltalk memory manager. Pointers to data that reside in this memory space can be passed freely to your C code. Your application is responsible for allocating and deallocating memory blocks located in this space.

Since the memory allocated by dynamic-link libraries is owned and controlled by the library, the Smalltalk image snapshot routines do not save the state of a dynamic-link library. It is the responsibility of the application developer to perform the necessary operations during image save and restore operations. For further information, consult External Interfaces and Snapshots.

A Smalltalk application can also receive pointers from your C code. These pointers are packaged into C data proxy objects which enable your Smalltalk application to manipulate the C data using standard Smalltalk message expressions.

A general overview of this data proxy and memory space layout is depicted in the following diagram.

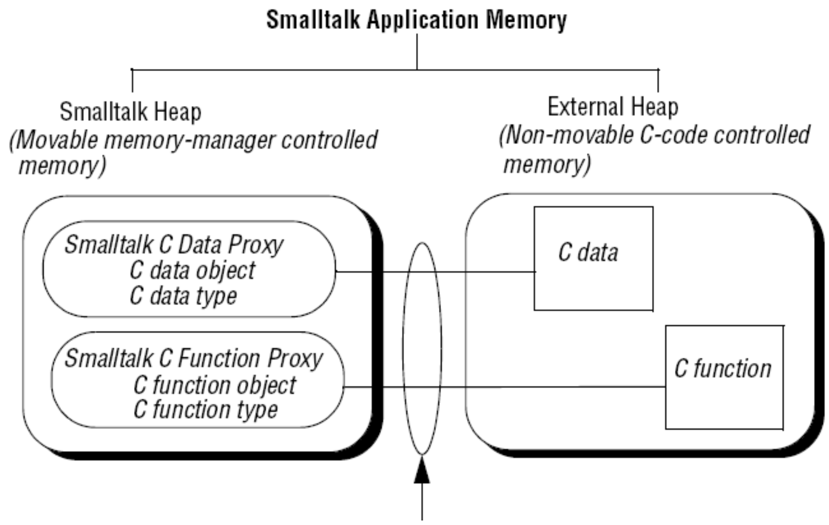

Smalltalk Application Memory
Smalltalk Heap
(Movable memory-manager controlled memory)
External Heap
(Non-movable C-code controlled memory)
Smalltalk C Data Proxy
C data object
C data type
Smalltalk C Function Proxy
C function object
C function type
C data
C function

Figure 2: Smalltalk application memory

Notice the following items in the above design:

- The Smalltalk C data proxy object contains a reference to the C data object that resides in the external heap. Your Smalltalk application can access the C data object through the data proxy object.
- The Smalltalk C data proxy object contains a type object. The proxy object knows the size and layout of the C data object from the data proxy object.
- The Smalltalk C data proxy object's external heap reference is broken on a return from snapshot. The external heap does not survive across snapshots, so all external heap memory must be reconstructed after every return-from-snapshot.

In addition to enabling your Smalltalk application to access code written in C, DLL and C Connect enables your C code to manipulate Smalltalk objects. This includes creating new object instances, accessing and setting the fields of an object, or sending messages to objects. This is done by using the Object Engine Access protocol, which is a set of C language functions. This interface is described in the chapter Object Engine Access Functions.

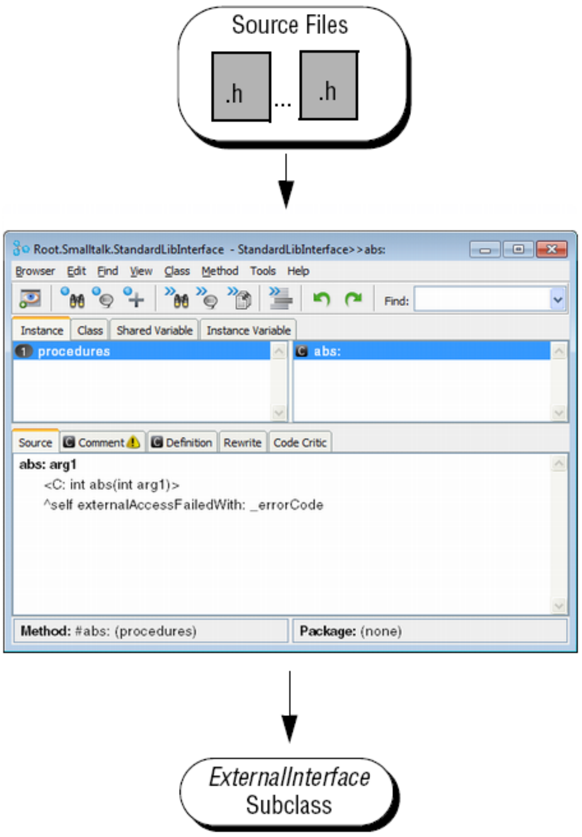
Finally, DLL and C Connect enables you to invoke C code that is either statically linked to the Smalltalk executable, or dynamically loaded using the platform's dynamic-link library facility. Instructions for creating and using dynamic link libraries on various platforms are provided in the chapter Platform Specific Information. Both static and dynamic linking are available on all supported platforms.

Constructing the External Interface

The process of developing an interface can be broken into two stages: first, the creation of Smalltalk ExternalInterface classes, and second, C data type and function prototype creation. To create an ExternalInterface you need access to the C module's Application Programming Interface (API). The API typically consists of a set of C language type declarations that specify the data types and function prototypes implemented by the C code. Use these C language declarations to build your Smalltalk ExternalInterface class. These declarations enable your interface to correctly communicate with the C code. Defining and using these C data types and function prototypes is described in greater detail in following chapters. For the time being, assume that you have access to these declarations, either in a manual or in a C language header file.

Starting with the C code's API, the Smalltalk interface can be created with the following two techniques:

- The first technique is to create the ExternalInterface class and its methods manually by using the standard Smalltalk source browsing and compilation tools. This technique is similar to programming in the Smalltalk development environment (VisualWorks), and involves manually entering the Smalltalk methods that represent the C language interface to the external C module. For each function or variable you want to access in the C code API, you must define a corresponding method in your interface class.


Source Files
.h
...
.h
Root.Smalltalk.StandardLibInterface - StandardLibInterface>>abs:
Browser Edit Find View Class Method Tools Help
Find:
Instance Class Shared Variable Instance Variable
procedures
abs:
Source Comment Definition Rewrite Code Critic
ExternalInterface Subclass

```
abs: arg1
    <C: int abs(int arg1)>
    ^self externalAccessFailedWith: _errorCode
```

Method: #abs: (procedures)   Package: (none)

Figure 3: Creating the ExternalInterface class

- The second technique is to use the automated External Interface Builder tool to generate the ExternalInterface class. This tool parses a C language header (`.h`) file, and builds the interface class using the C language definitions. Every C language construct in the header file is compiled into an associated Smalltalk method in the interface class. This technique is faster, but may result in larger interface classes if your interface header file includes other header files (typically system supplied header files). Use this automatic facility with care.

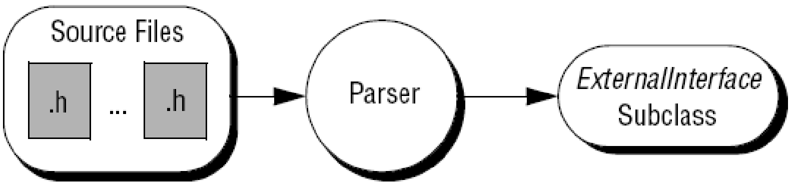


Figure 4: Generating the ExternalInterface class

C code modules are platform dependent, but typically consist of one of the following code objects:

- Third-party dynamic-link libraries
- User generated dynamic-link libraries
- C code statically linked to your Smalltalk Object Engine (either third-party code or your own code)

The process of building dynamic-link libraries and statically linked Object Engines is summarized in the following diagram, which illustrates the various possible paths when designing and building interface classes.

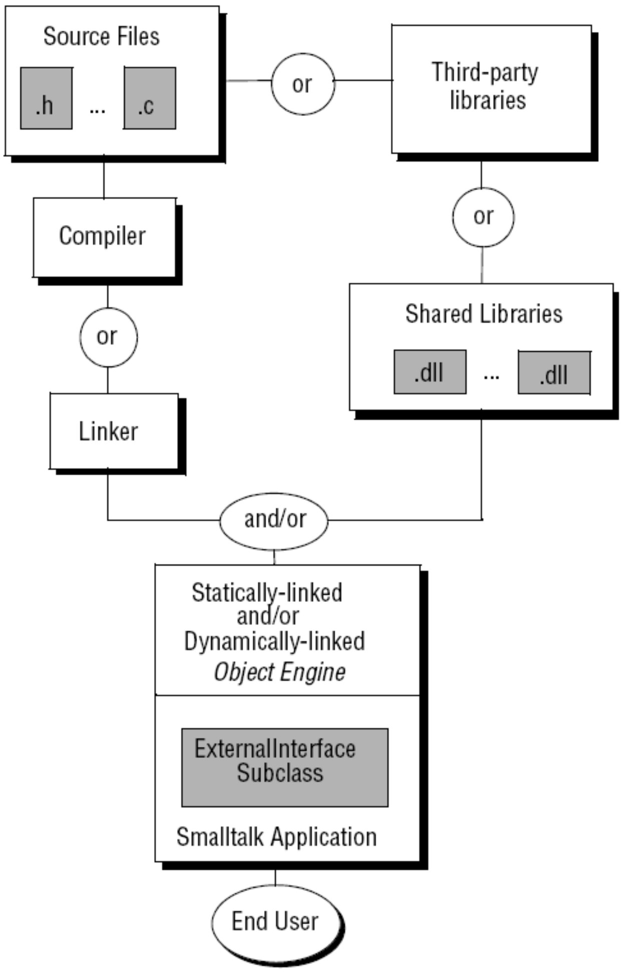


Figure 5: Building dynamic-link libraries and statically linked Object Engines

Conceptually, DLL and C Connect is divided into two parts. The first part is a group of development tools. These are used only while developing an application that makes use of the `ExternalInterface` functionality. The second part is a collection of classes known as the run-time or support classes. These classes are used while your application is running and must be present in the base system.

User Interface Tools

DLL and C Connect provides two special development tools for constructing interfaces to external code: the Finder and the Builder. These tools are used specifically for automating the task of building and integrating an ExternalInterface into your Smalltalk application, and are thus designed as extensions to the standard VisualWorks development tool set. These special tools are the following:

- The External Interface Finder is a special browser that enables you to view a group of different ExternalInterface classes.
- The External Interface Builder is a more specialized tool that provides you with greater control over the process of automatic interface construction.

In this section, the Finder and the Builder tools will be described roughly in the order in which they might be used in a typical project.

Accessing the Tools

To begin using the DLL and C Connect user interface tools, perform one of the following:

- In the VisualWorks main window, choose **Tools > DLL and C Connect.**
- Click the DLL and C Connect icon in the VisualWorks Launcher window.

The External Interface Finder tool opens. You may now begin using the tools to build or manipulate external interfaces.

External Interface Finder Tool

To use this tool, load the DLLCC-Tools parcel.

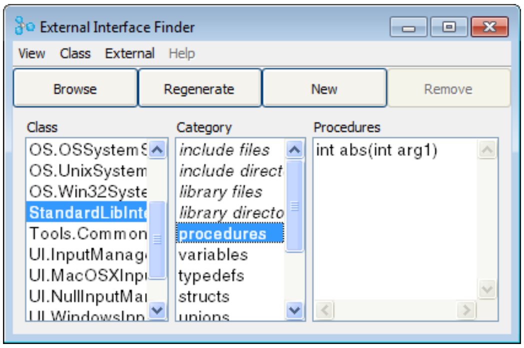


Figure 6: Finder Tool

The External Interface Finder enables you to navigate easily among a group of ExternalInterface classes. Individual ExternalInterface classes are listed hierarchically in the main view.

The buttons above the list views in the Finder enable you to Browse the code for a selected interface, Regenerate the external methods, add a New interface, and Remove an existing interface class. Note that double-clicking on a class name in the list view has no effect.

Building an Example: StandardLibInterface

The following steps describe how you can build an example ExternalInterface class which can be used to make calls on the standard C library known as `stdlib`. This example presupposes that you have access to a dynamic-link library (DLL) version of `stdlib` available on your development workstation. Note that each platform will use different conventions to identify this library (for example, on Windows, these libraries are identified with the `.dll` extension).

Perform the following steps to create a new class, StandardLibInterface. In this example, we will define the class and then add a method for calling the standard C function abs(), which performs the absolute value mathematical function.

1. Within the Finder tool, add a new ExternalInterface class by pressing the **New** button.
2. In the interface definition dialog, enter the name of the new class (StandardLibInterface) in the **Name** field, select a name space (e.g., Smalltalk), and click **OK**.

The new class and a list of categories appear in the Finder. The external objects that are available are procedures, variables, typedefs, structs, unions, enums, defines, and macros.

3. To add a new procedure, select the external category, **procedures** in the column under **Category**, and press the **New** button.

   A dialog box appears that prompts for the information required to define a well-formed external declaration. The dialog box has additional selection buttons depending on the external's category.

4. In the procedure dialog box, enter abs (absolute value) as the name of the procedure.
5. Choose the **int** menu option as the return type.

   Instead of choosing one of the menu options, you may also enter a return type that is not included as an option.

6. Select **argument 1** under the list of argument types.
7. Choose the **int** menu option, to correspond to **argument 1**.

   Additional arguments are defined by selecting the consecutive arguments, and choosing their corresponding argument types. The type you select for each argument will be stored as you move down the list.

8. Click **OK**.

Clicking on **OK & Browse** adds the new procedure and brings up a browser on the new method. You can add to or change the procedure declaration from the browser.

You have now defined a method in the class `StandardLibInterface` that will enable you to call the C function `abs()`. To complete the interface, you must next add some additional information about the external C library `stdlib`. To define external objects for the interface, you first select one of the italicized external categories (include files, include directories, library files, or library directories).

Testing the Example: StandardLibInterface

By following the steps in the previous section, you have constructed a new class `StandardLibInterface` that contains methods for calling C functions in the library `stdlib`. All that is needed now to actually make a call to this library from Smalltalk is a reference from the interface class to the library DLL.

The name and location of this library file varies depending upon the platform that you are using. On Windows, the C library is one of the `MSVCRTnn.DLL` files, while on Solaris the C library is `/usr/lib/libc.so`. You can also make use of environment variables to simplify the path specifier to the `stdlib` library DLL on your platform (for details, see: Dynamic-Link Libraries. If you are unsure how to locate the `stdlib` library DLL on your system, consult the documentation for the C development environment supported by your platform.

For example, assuming that the target platform is running Windows NT 4.0, we would set up the libraries for `StandardLibInterface` as follows:

1. Return to the External Interface Finder tool and select `StandardLibInterface` in the list of interface classes (the left-most pane in the view).
2. Select **library files**, under the Category list, and press the **New** button.
3. A dialog box appears. Enter the name of the library file (for example, on Windows NT 4.0, enter **msvcrt40.dll**) and click **OK**.
4. Select the external category, **library directories**, and press the **New** button.
5. Enter the directory path where the library file is located and click **OK**. You can make use of an environment variable here to simplify the path (for example, on NT platforms, the path of this directory is either `$(windir)\system` or `$(windir)\system32`). You can specify both, and the system will search for the library file.

The external interface class has now been associated with the specified C library, and the library will be loaded and linked dynamically as soon as you call the function. To test this, you may execute the following Smalltalk expression in a workspace. Select the code and choose **Print It** from the <Operate> menu to see the returned value. Execute it a second time to see that it runs much faster after the linking step has been performed once:

```
StandardLibInterface new abs: -10.
```

The result should be: `10`.

Evaluating the previous expression may result in a notifier that indicates the entry point for `abs()` could not be found. In this example, because `abs()` resides in a dynamic-link library, the notifier may indicate that the library could not be found or loaded. In either case, verify that the library name and library paths specified in the `ExternalInterface` class definition are correct and that a correct version of the library actually resides in the specified location. To troubleshoot any exceptions raised during the external call, see: Resolving Exceptions.

The procedure for building an `ExternalInterface` described in this section requires that you define the interface methods by hand. DLL and C Connect also provides a more sophisticated mechanism, the External Interface Builder tool, for automatically generating your interface class.

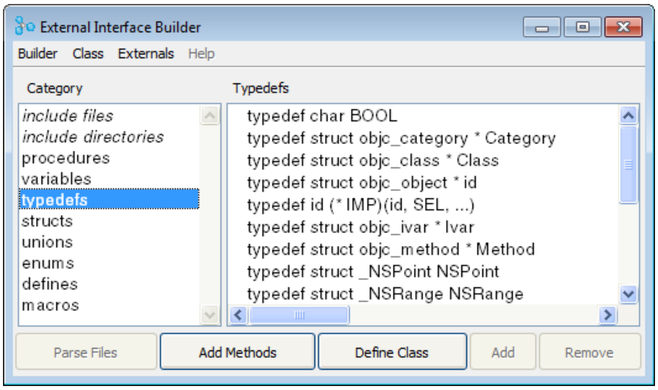
External Interface Builder Tool

To use this tool, load the DLLCC-Tools parcel.

To construct an `ExternalInterface` class from declarations in a C language header file, DLL and C Connect provides a special tool for automatically parsing the declarations in an interface file and then compiling the corresponding methods for your `ExternalInterface` class. The Builder tool provides a means to accomplish these two tasks (parsing and compiling).

Once the Builder has parsed all the header files that define a given C interface, you may select only those declarations from the parsed set that you wish to include in your interface class. Thus, by using the Builder tool, you may parse a group of header files and then selectively construct an interface class.

To open the Builder from the External Interface Finder, choose the **Class > Builder** menu item in the Finder. The External Interface Builder tool opens on your screen.



include files
include directories
procedures
variables
typedefs
structs
unions
enums
defines
macros

```
typedef char BOOL
typedef struct objc_category * Category
typedef struct objc_class * Class
typedef struct objc_object * id
typedef id (* IMP)(id, SEL, ...)
typedef struct objc_ivar * Ivar
typedef struct objc_method * Method
typedef struct _NSPoint NSPoint
typedef struct _NSRange NSRange
```

Figure 7: Builder Tool

The Builder tool contains an input field to specify the header files to parse and an input field indicating the directories that contain the named header files (i.e., the paths to the header files). When you have entered the name(s) and paths of the header file(s), the tool is ready to parse them. Note that only the declarations you select will be compiled. The tool will then automatically generate the selected methods in your interface class.

This tool is especially useful in situations where you are presented with a large header file (for example, Motif's `Xm.h` header file). This tool enables you to parse the header file and extract only the declarations you require. It is intelligent enough to automatically mark dependent types for inclusion in your interface class.

Note that the Builder was written to parse ANSI format source code.

The buttons located on the bottom of the Builder tool enable you to parse header files, add methods to a class, define a new class, add new external categories, and remove external categories.

Continuing with our example from the previous section, we can now automatically construct an interface to the standard C library `stdlib`. You may include some or all of the functions from this library by first parsing the header file `stdlib.h`. To parse this header file, perform the following steps using the Builder Tool:

1. In the list of external protocols, select the category **include files,** and press the **Add** button.

2. A dialog box appears. Enter the name of the include file (for this example, `stdlib.h`) and click **OK**.
3. Select the external category, **include directories**, and press the **Add** button.
4. Enter the directory path where the include file is located and click **OK** (for example, on UNIX platforms, the path of this directory is usually `/usr/include`, while on Windows machines it is usually `C:\MSDEV\include\`). For additional details on path specifiers, see: Dynamic-Link Libraries.
5. By defining the include files and include directories variables, you have set up your `ExternalInterface` to automatically parse a C interface. To actually perform the parsing operation, press the **Parse Files** button.

The files and directories are parsed top to bottom. If more than one directory is listed, the parser looks at the first directory. If the header file is found in the first directory, the file is parsed from that directory. If it is not found, the parser proceeds to the next directory, until a match is found. In this manner, you can define interfaces that can be parsed on platforms with different path or name specifiers for the external libraries.

Once the header file has been parsed, you can choose particular methods simply by selecting them individually on the list of function definitions. As you select them, a check mark will appear in the left margin of the list view.

For example, suppose you want to add the methods `atoi()` and `atof()` to be included in the `StandardLibInterface` class.

1. Select **procedures**, under the Category list.
2. Select `int atoi(const char *)` and `double atof(const char *)` under the **Procedures** list. A check mark appears next to the selected methods.

   You can select all the procedures in the header file by choosing the **Externals > Select All** menu option (located on the top of the parser window). Conversely, you can also deselect all of the procedures using the **Externals** menu.
3. For now, add only the selected methods `atoi()` and `atof()` to the external class `StandardLibInterface`. Press the **Add Methods** button. The

default class name (StandardLibInterface) appears in the dialog box. Click **OK**.

When you next open the Finder tool, it will display the added methods. Note that if the Finder is already displayed, you may need to perform **View > Update** in the Finder.

Since the external library has already been specified (see: Building an Example: StandardLibInterface), you may now test these function calls. For example, to test the atoi(), execute the following code fragment:

```
StandardLibInterface new atoi: '12.05'.
```

Since the C library function atoi() takes an ASCII string and returns a C int type, the result is converted into the SmallInteger 12.

These simple examples demonstrate some of the principal aspects of DLL and C Connect's ExternalInterface class. In the next chapter, we will look more closely at the protocol of this class, and how you can define your own interface classes.

Chapter

2

Defining External Interfaces

Topics

This chapter explains how to define your own External Interface classes, how to declare and call C functions, how C header files are parsed, and how to access external variables defined in C libraries. It also explains in more detail how to link your interface classes with the appropriate external libraries, and how you can also call external code modules written in languages other than C.

Defining Interfaces

Defining and using External Interfaces is a three-stage process:

1. Create an interface class with Smalltalk methods that parallels the declarations in the external code's C header files or from the external code's programming interface manual. Parsing C header files is typically done automatically.
2. Register the C code modules.
3. In your Smalltalk application, invoke the C functions.

In the previous chapter, the External Interface Finder tool was used to create a new interface class (the example StandardLibInterface). Sometimes, however, it is easier or necessary to create a new ExternalInterface class using the System Browser. For example, if no C header file exists, this approach may be necessary.

To create a new interface class using the Browser, first select the name space and class category in which you want to include the new interface. In the example that follows, we shall assume the category name is ExternalInterface-New. If a class in that category is already selected, deselect it.

Starting with the standard class definition template, remove and replace the dummy names, as in the following finished example:

```
Smalltalk defineClass: #ExampleInterface
 superclass: #{External.ExternalInterface}
 indexedType: #none
 private: false
 instanceVariableNames: ''
 classInstanceVariableNames: ''
 imports: ''
 category: 'ExternalInterface-New'
```

After you are finished, select **Accept** from the code view's <Operate> menu to display the class name. Notice that the class definition template has changed. Several new elements now appear in the class creation message of the code view:

```
Smalltalk defineClass: #ExampleInterface
 superclass: #{External.ExternalInterface}
 indexedType: #none
private: false
 instanceVariableNames: ''
 classInstanceVariableNames: ''
 imports: '
   private Smalltalk.ExampleInterfaceExternalDictionary.*
   '
 category: 'ExternalInterface-New'
 attributes: #(
   #(#includeFiles #())
   #(#includeDirectories #())
   #(#libraryFiles #())
   #(#libraryDirectories #())
   #(#beVirtual #false)
   #(#optimizationLevel #debug))
```

The above example is a typical `ExternalInterface` subclass creation message. The `attributes:` argument provides information specific to external interface classes. When subclassing `ExternalInterface` with the standard class creation template, these attributes are given the default values shown above. The attributes are as follows:

includeFiles

The `Array` argument is a sequence of `Strings`, each being the file name of a C language header file associated with the external interface, for example: `#('stdio.h')`. The `ExternalInterface` class enables you to automatically parse the listed header files and generate the corresponding Smalltalk methods. The process of parsing header files is explained in the discussion of Parsing C Header Files.

includeDirectories

The `Array` argument is a sequence of `Strings`, each being the file system directory name that is searched when attempting to locate the include files listed in `includeFiles`. The directories are searched left to right in the list, so ordering is important.

libraryFiles

The `Array` argument is a sequence of `Strings`, each being the filename of a dynamically loadable library (DLL) associated with the interface. Each platform has different naming conventions for libraries. For details, see: Platform Specific

Information. Optionally, each filename string can also be matched against the current platform in order to create a cross-platform interface definition. For details, see: Dynamic-Link Libraries.

libraryDirectories

The Array argument is a sequence of Strings, each being a directory in the file system that will be searched when attempting to locate a library file listed in libraryFiles. The directories are searched left to right in the list, so ordering is important. Directory names can also include environment variables. For details, see: Dynamic-Link Libraries.

beVirtual

The argument is a Boolean value. A value of true indicates the new interface class is a virtual class. A value of false indicates it is a normal class. For more details, see: Virtual External Interfaces.

optimizationLevel

The argument is either the Symbol #debug or #full, which indicates the level of optimization used when compiling each function method in the interface. In #debug mode, function methods contain strict type-checking wrapper code. This type-checking code helps in the development and debugging of your interface class at the expense of performance. With a #full optimization level, the type-checking wrappers are removed from the function methods, resulting in a significant decrease in function call overhead.

The rest of the ExampleInterface class definition behaves the same as the standard subclass definition template. You can use the normal class editing tools to add class comments, rename the class, delete the class, move the class to a new category, or file the class in and out of the system. Class ExampleInterface also provides protocol for changing these attributes at run-time.

Note that when you **Accept** the ExampleInterface class definition, a namespace called ExampleInterfaceDictionary is created and a private general import to its contents is added to the imports. Its purpose is to act as a repository for special interface objects created during the compilation of your C language interface objects. You need not be concerned with the contents or format of this dictionary.

To look in more detail at a working interface class, use the Browser to select the example class `StandardLibInterface`, which was described in the previous chapter. The `libraryFiles` and `libraryDirectories` attributes of the class specify the library (or libraries) used by this interface. Each platform names these libraries differently. In addition, each platform has its own mechanism for creating a dynamic-link library. In the example described previously, the Windows library **`msvcrt40.dll`** was registered as the library containing functions for `stdlib` (each platform uses different library file extensions).

Using the System Browser, you can edit these library definitions. For example, if you wanted to associate another DLL file with the interface, you would simply change the `libraryFiles` and `libraryDirectories` attributes of the class definition. Re-accepting the class definition using the code view's <Operate> menu recompiles the `StandardLibInterface` class, registering the new library file(s) with the class.

You can also edit method definitions directly using the code view in the System Browser. Function call method definitions are contained in the category named procedures.

For example, you can add the `atol()` function prototype definition within your interface class. As defined in **`stdlib`**, this function converts a C string into a long integer. Make sure the `StandardLibInterface` class is selected in the class view of your browser, and within the procedures category, type the following method definition:

```
atol: aString
 <C: long atol(const char * )>
```

When you are satisfied with the method definition, select **Accept** in the code view's <Operate> menu.

Since the **`stdlib`** dynamic-link library is already associated with the example class `StandardLibInterface`, you can immediately execute the following Smalltalk expression in a workspace. Select the code and choose **Print It** from the <Operate> menu to see the returned value.

```
StandardLibInterface new atol: '98765432'
```

The result of this expression should be a `LongInteger`.

In most cases, you will be linking pre-existing C library and header files, so the procedures you just followed can be abbreviated. Simply create an interface class and then send messages to that class from your Smalltalk application.

Defining External Methods

To define interface methods for C declarations, create a Smalltalk method in the following form:

```
declarationName
 "comment"
 <C: declaration>
 failure code
```

The syntax for Smalltalk external methods is as follows:

- Except for the identifier, a `C:` instead of a `primitive:` appears enclosed in the angle brackets.
- The C declaration appears after the colon, and specifies one of the following C declarations: typedefs, function prototypes, defines, macros, and variables.
- `failure code` is a set of Smalltalk expressions for handling function call or global variable access failure situations.

Declaring C Data Types

DLL and C Connect provides a type-matching facility that equates standard C data types with equivalent Smalltalk classes. `CType` and its subclasses represent C types such as integers, floating-point numbers, pointers, arrays, structures, unions, enumerations, procedures, and voids.

All `CType` classes provide various instance-creation methods to create instances of the `CType` objects, which represent the actual C types.

For example, the expression `CIntegerType char` returns an instance of `CIntegerType` that represents the actual C `char` type. The expression `CIntegertype char pointerType` returns a `CPointerType` object that represents the C `char *` type. In general, you do not have to deal with these classes directly.

The standard C data types are built into the product and conversions to these types from Smalltalk objects are handled automatically. Each custom data type that you define using a `typedef` expression in your C code must have a corresponding accessing method in the C code's interface class. As with function declarations, a data type declaration consists of the C `typedef` statement prefaced by `C:` and enclosed in angle brackets. For example, suppose the `size_t` type is declared in a C header file as follows:

```
typedef int size_t;
```

The equivalent declarative method in the interface class is as follows:

```
size_t
 <C: typedef int size_t>
```

DLL and C Connect constructs these methods automatically for all `typedef` statements in the header files that are named in the `includeFiles` attribute of the interface class definition. To further restrict the list of included declarations, use the `generateMethods:` protocol in `ExternalInterface class`.

When you send the `size_t` message to your interface class, it answers an instance of a `CTypedefType`, a subclass of the base abstract class `CType` described previously. In general, you will only need the memory allocation protocol that all `CType` objects understand. Use this protocol to allocate C data objects of the given type.

Note: DLL and C Connect provides several strategies for managing C data objects passed to external libraries. It is absolutely critical that you choose the strategy that is appropriate for the allocation behavior of both your application and the functions in the external library. A complete discussion of the various C type objects and appropriate memory allocation strategies appears in the discussion of Creating and Accessing C Data. For multi-threaded applications, also see: Managing Data Objects with Multiple I/O Threads.

In the following section, we'll look more closely at how C functions are called from Smalltalk methods.

Declaring enums

DLL and C Connect provides support for enumeration types. Enumeration types are symbolic names with integer values. These are defined in essentially the same way as data type declarations, with a method that contains a pragma. For example, the `months` type might be declared in a C header file as follows:

```
enum months{ Jan, Feb, Mar, Oct = 10};
```

The equivalent declarative method in the interface class is:

```
fewMonths
 <C: enum months{ Jan, Feb, Mar, Oct = 10}>
```

This method returns a `CEnumerationType` object containing the symbolic values. DLL and C Connect constructs these methods automatically for all `enum` statements in the header files that are named in the `includeFiles` attribute of the interface class definition.

For details on accessing enumeration types, see: Enumeration Types.

Declaring C Functions

For each C function that you use, create a Smalltalk method that takes the appropriate number of arguments and defines the function prototype. The function prototype is in the ANSI C prototype format. It is also prefaced by `C:` and enclosed in angle-brackets, so the Smalltalk compiler can recognize it as a C declaration. Returning to the `StandardLibInterface` example, recall that the header file contained a prototype for the `abs()` function. After defining the `StandardLibInterface` class, the following Smalltalk method was also defined:

```
abs: arg
```

```
<C: int abs(int)>
```

The method, like the function, takes one integer argument. The following example takes two string arguments (note: this function is defined in `string.h`, not `stdlib.h`). It calls the C `strcmp` (string comparing) function:

```
strcmp: s1 with: s2
 <C: int strcmp(const char *s1, const char *s2)>
```

Looking at the two example methods given above, notice that argument names in the declaration are included in the second example, but not in the first, illustrating their optional nature. However, note that in any given method you must either include all of the argument names or omit all of them.

When arguments are passed, an ellipsis can be used in the declaration, but all of the function's arguments must be passed in an array. This holds true even if you specify arguments before the ellipsis. For example, consider this declaration of the `printf` function (C printing function):

```
printf: argArray
 <C: void printf(const char *, ... )>
```

The argument to `printf:` must be a Smalltalk `Array`. The first element of the array is a Smalltalk `String` object (or a `char *` C pointer object) and the subsequent array elements are the arguments as expected by the `printf()` function based on the string encoding of the first argument. Use this array as the argument to `printf:`.

As mentioned previously, declaration methods can be created automatically after you specify one or more header files in the definition of the interface class. You can also create them manually, using the System Browser as you would for an ordinary Smalltalk method. You might prefer that approach when no C header file exists, and you have no other reason to create one. You might also want to rename the keywords and arguments in the method selector to give them a Smalltalk flavor rather than the parser's C flavor. For example, the strcmp method might be written as follows:

```
compare: string1 to: string2
 <C: int strcmp(const char *string1, const char *string2)>
```

You can assign the result of a function into an instance variable within the brackets, using an equal sign. For example, the following code shows how to store the result of atol() in an instance variable named result:

```
atol: aString
 <C: result = long atol(const char * )>
```

Any variable that exists within the scope of the method can be used as an argument in a declaration; this includes instance or temporary variables.

Note: There is an important limitation with the scheme used by DLL and C Connect to represent C types. In particular, if you plan to define a number of different interface classes, using the Smalltalk class hierarchy to distinguish between "abstract" and "concrete" interface definitions, then you should read the discussion concerning Limitations of CType Definitions.

Calling C Functions

After you declare the necessary functions and link the library, your interface class is ready to invoke external functions. To do so, create an instance of the interface class and send it messages from its procedures message category. These methods contain function-prototype declarations as described in the previous section of this chapter. When you generate an interface automatically by parsing a header file, the parser places all procedure methods in the procedures category.

In the StandardLibInterface example described previously, the atol() function was invoked by sending an atol: message to an instance of StandardLibInterface:

```
StandardLibInterface new atol: '98765432'
```

C Function Failure

In a manner similar to Smalltalk primitive methods, failure code can follow a method’s C procedure prototype declaration. The failure code, which consists of Smalltalk expressions, is only invoked when the C function fails to return a value. An example of a method with failure code is as follows:

```
atol: aString
 <C: long atol(const char * )>
 ^aString asNumber
```

In general, however, you will probably want to have error handling code follow the C function call. Class `ExternalInterface` also provides a standard error handling protocol which provides a framework for resolving C function call failures. The cause of the function call failure is made available to you for more sophisticated error handling. Thus, a more typical example would look like this:

```
atol: aString
 <C: long atol(const char * )>
 ^self externalAccessFailedWith: _errorCode
```

When the C function call fails, the cause of failure is stored in `_errorCode`, which is a special temporary variable in the method containing the C function prototype. This hidden temporary variable holds an instance of the class `SystemError` which indicates the reason for the failure. By convention, the method `ExternalInterface>>externalAccessFailedWith:` is invoked to raise an `externalAccessFailedSignal.` In the code fragment shown above, the signal will be raised with the `SystemError` object that contains the exact cause of the failure.

The type of error is stored in the `name` field of the `SystemError` object, and this field is usually tested by the error handler. There are a number of standard errors that can occur during a C function call failure, and your application must provide an exception handling mechanism to recover from these errors. A more detailed discussion of C function call failures, as well as a general discussion of exceptions, is located in the discussion of Exception Handling.

Declaring Defines, Macros and Pragmas

DLL and C Connect can parse and generate methods for #define statements. Unfortunately, many #define statements in C are complex expressions, or source code that requires both a run-time parser and context in which to evaluate the #define.

Note: DLL and C Connect imposes a run-time restriction that any interface method representing a #define returns the value nil, unless the #define represents a scalar or string object.

An example of a #define statement and its associated Smalltalk method is as follows. The C define statement is:

```
#define ARRAY_SIZE 100
```

The corresponding Smalltalk method is:

```
ARRAY_SIZE
 <C: #define ARRAY_SIZE 100
 >
```

Note that a carriage return is embedded inside the angle brackets. This is required because the #define statement is terminated by a newline. It is important to note that the statement will not parse correctly without this newline character.

This restriction in the parsing of #define statements will be removed in a future release of DLL and C Connect.

Example expressions that are possible in the body of a #define, and that are correctly parsed into a Smalltalk object are as follows.

Table 1: Example expressions

Example expression	Evaluates to the class	Whose value is
#define CHAR 'c'	Number	98
#define MULTI_CHAR 'abcd'	Number	0x61626364
#define STRING "aString"	String	'aString'
#define EXPRESSION (1 << 3)	Number	8

For details on arithmetic operators supported by the last type of #define expression listed above (#define EXPRESSION), see: #define Operators.

Define statements that accept arguments (macros) are only supported in a development image (an image where the DLL and C Connect classes are loaded). The evaluation of a macro will answer nil if it does not reduce to a scalar or string value.

An example of a #define macro statement and its associated Smalltalk method is shown below. The example macro computes the number of bytes required to represent a multi-element data structure. Note that a carriage return is embedded inside the angle brackets used in the Smalltalk method. This is required because the #define statement is terminated by a newline. The C define statement is:

```
#define NUM_BYTES(type, nElem) (sizeof(type) * nElem)
```

The corresponding Smalltalk method is:

```
NUM_BYTES: type with: nElem
 <C: #define NUM_BYTES(type, nElem) (sizeof(type) * nElem)
 >
```

The following code snippet is an example of evaluating the macro. The first argument is the C long type and the second argument is the number of elements. Note that the arguments are String objects. Evaluating the expression on a platform whose long data type is 4 bytes will answer the SmallInteger 20.

```
InterfaceClass new NUM_BYTES: 'long' with: '5'
```

Various compilers implement compiler-specific #pragma directives. The parser correctly parses these pragma directives. However, they are not used for any purpose and are discarded. For more information on #pragma directives, consult your compiler's reference manual.

Declaring Variables

In addition to function entry points, C libraries can export global variables. To retrieve and set the value of a global variable, you must first define two Smalltalk methods in your interface class. The first method takes no arguments and is used to return the variable's value. The second method accepts one argument and is used to set the variable's value. When you generate methods using the automatic parsing mechanism to build your ExternalInterface class, it always generates both accessing methods.

The C language type qualifier extern is optional, and it is simply discarded by the parser. For additional details on the parsing of C header files, see: Parsing C Header Files.

The following example methods are used to set and retrieve the value of a global variable, in this case globalVariable, that is defined in a library or statically linked to the Object Engine.

```
globalVariable
 <C: extern int globalVariable>

globalVariable: value
 <C: extern int globalVariable>
```

External Variable Failure

When attempting to access an external variable that is located in a linked library (i.e., a static, global variable), DLL and C Connect may raise one of the three following exceptions:

Cannot load library

This error may occur when your application first attempts to access an external variable. The variable is accessed by loading the library that your external interface class indicates contains the variable. If the library could not be loaded for any reason, the signal ExternalLibrary>>libraryNotLoadedSignal is raised.

Cannot find library

This error may occur when DLL and C Connect cannot find the library specified in your interface's class creation template. To indicate this failure, the signal ExternalLibraryHolder>>libraryNotFoundSignal is raised.

Cannot find external object entry point

If DLL and C Connect successfully loaded the library associated with your interface class, but subsequently could not find the external variable within the library or as a statically linked object. The signal ExternalMethod>>externalObjectNotFoundSignal is raised.

To catch these signals, you can write signal handlers in the callers of your interface methods. For a more detailed discussion of these errors, see: Exception Handling.

Virtual External Interfaces

ExternalInterface classes contain a special attribute that can be helpful if you want to link multiple versions of the same C function library to your Smalltalk application. The virtual attribute controls the way that method lookup is performed in a hierarchy of ExternalInterface classes.

A subclass inherits the includeDirectories, libraryFiles, and libraryDirectories attributes of its parent interface class. Function look-up begins in the receiver's libraries and continues up the hierarchy. For example, you could create OracleInterface and SybaseInterface as subclasses of DatabaseInterface. Selected functions in the parent's directories could then be overridden by more specific functions in the subclass directories.

The following diagram illustrates this scheme:

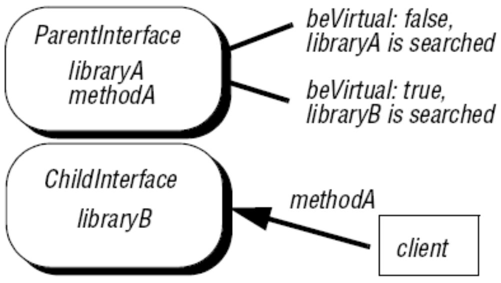

ParentInterface
libraryA
methodA
beVirtual: false, libraryA is searched
beVirtual: true, libraryB is searched
ChildInterface
libraryB
methodA
client

Figure 8: Inheritance

A subclass does not inherit includeFiles, though the associated methods are inherited by the normal method-inheritance mechanism. When an inherited method is invoked, the function look-up order is affected by whether the parent is virtual or not. With a nonvirtual parent, an inherited method causes the function look-up to begin in the parent class's libraries. That is, in the libraries associated with the class that implements the method. With a virtual parent, the function look-up begins in the message receiver's libraries. In effect, the subclass of a virtual parent can substitute an alternate version of the parent's libraries, and need not re-implement the accessing methods.

An interface class can be made virtual in two ways:

- Send `beVirtual: true` to the class
- Recompile the interface class, specifying `beVirtual: true` in the class-creation template

If you plan to define a number of different interface classes, using the Smalltalk class hierarchy, then you should read the discussion of Limitations of CType Definitions.

External Interfaces and Snapshots

Smalltalk preserves the semantics of objects across a snapshot in most cases. The means to do this are provided by the snapshot facility itself, which preserves the state of the object memory. For objects whose interpretation is completely contained in an image, this is sufficient.

However, some types of objects are affected by the external environment and must be given special treatment. In particular, references to external libraries that were active when the snapshot was made will become invalid. When a snapshot is made, all references to dynamically loaded libraries are broken.

When an image is restarted, all external interfaces will, on demand, reload libraries listed in the external interface's `libraryFiles` attribute. If there is a possibility that the image file will be moved to a new environment, or if the library files might be moved, your application should re-install the new location of the library files using the `libraryFiles:` and `libraryDirectories:` messages available to the `ExternalInterface` class. This is fully described in the discussion of Finding Entry Points.

The `ExternalInterface` class provides two mechanisms for coping with snapshot returns:

- Class ExternalInterface sends a returnFromSnapshot message to each of its subclasses, informing the class that the image is starting, potentially on a new platform. Subclasses can override the default behavior to perform platform-specific initialization.
- The class also detects the current platform.

When Smalltalk returns from a snapshot, it performs the following sequence of events:

1. When the image first starts, the class ObjectMemory receives the message returnFromSnapshot. That method performs some system initialization and sends the change message earlySystemInstallation to itself.
2. Any object that registered itself as a dependent of ObjectMemory is notified of the earlySystemInstallation. ExternalInterface is one class that is registered as a dependent of ObjectMemory. Note that if you use the earlySystemInstallation notification, the window system will not have been correctly installed, so notifiers will not work. Unhandled errors may crash the system without warning.
3. ObjectMemory performs more installation, then sends a returnFromSnapshot change message to itself. Any object that registered itself as a dependent of ObjectMemory will be notified of the returnFromSnapshot.

To provide a hook for its subclasses, ExternalInterface is registered as a dependent of ObjectMemory. When ObjectMemory sends the earlySystemInstallation change message, ExternalInterface performs its own initialization, sending the installOn: message to each of its subclasses.

By default, installOn: flushes any cached information in external methods. You may override installOn:, but be sure to use super installOn: as the first line of your method. This ensures that external caches get flushed, ensuring an easier migration to future implementations.

The argument to installOn: is a two-element array. The first element is a Symbol that indicates the platform class your application is currently running on (for example, #win32, #mac, #os2, or #unix). The second argument is a string that describes details of the current platform. For example, the argument to installOn: for a MS-Windows NT platform would be #(#win32 'win32 V4.0 nt i386'). The second argument varies depending on the current configuration of your machine.

The argument to installOn: is available anytime during the execution of your application by sending the message currentPlatform to the class ExternalInterface, or any of its subclasses.

A list of platforms and their currentPlatform array is shown in the following table. Note that the second array argument is dynamically generated on some platforms based on the current operating environment. Note that these values may vary depending on the actual platform configuration.

Table 2: Platform-specific currentPlatform arrays

Platform	Example identification
MS-Windows NT	#(#win32 'win32 V4.0 nt i386')
MS-Windows 2000	#(#win32 'win32 V5.0 nt i386')
MS-Windows XP	#(#win32 'win32 V5.1 nt i386')
OS/2	#(#os2 'os2 OS/2 V2.0')
Mac OS X	#(#unix 'unix bsd apple Power MacOSX')
IBM RS/6000	#(#unix 'unix bsd ibm rs6000 aix')
Solaris 2.5	#(#unix 'unix sysV sun solaris')

Other initializations occur when the image is first started and are discussed in: External Heap and Snapshots.

Dynamic-Link Libraries

Depending upon whether you choose static or dynamic linking of C modules, the strategies for loading the actual C code differs slightly. With dynamic linking, you only need to arrange for the library files to be loaded by DLL and C Connect. With static linking, the code is already linked into the Object Engine executable, but you need to register the functions that you call. Although it is recommended that you choose dynamic rather than static linking, DLL and C Connect does support both approaches. To register your statically linked files, use the special mechanism described in: Static Linking. The following discussion explains how to arrange for DLL and C Connect to load dynamic-link libraries.

Finding Entry Points

On a platform that supports dynamic-linking, a library is registered when you compile the interface class (by accepting the class definition). The library file is specified in your interface class by setting the file name in the `libraryFiles` class template attribute.

DLL and C Connect uses a lazy approach to library loading and linking. When an interface class needs to find an entry-point address for an exported symbol, it looks for that unbound entry point in the interface's `libraryFiles` list. It scans the library list in the left-to-right order you specified in the `libraryFiles` attribute. When it encounters a library that has been loaded, it simply performs an entry-point look-up. If the library has not been loaded, it first loads and links the library into memory, and then performs a lookup of the entry-point. If the entry-point is not found in the library, it continues the look-up with the next library in the list.

When the library list is exhausted, it is checked for entry-points statically linked and registered with the running Object Engine. To statically link and register an entry point, see: Static Linking. Because the look-up mechanism searches libraries before statically linked code, a dynamically linked function overrides a statically linked version.

If the search fails to find the entry-point in any library or the statically linked code, the exception `ExternalMethod externalObjectNotFoundSignal` is raised. Consult Exception Handling for a comprehensive discussion of exceptions your interface code may raise.

Once the look-up is complete and the correct entry-point has been identified, the function's address is cached so that future calls on the library routine will not require a complicated look-up. Thus, the speed of subsequent calls will be faster than the first call.

Library Search Order

In the interface class-creation template, you can specify multiple libraryFiles and libraryDirectories attributes. As the name suggests, libraryDirectories are the directories in which the libraryFiles are located.

Your interface class searches for a library file in the directories listed using a left-to-right search order. If the library file cannot be found in the first library, the search continues with the next library in the list. If the end of the list is reached and the library has not been found, an ExternalLibrary libraryNotLoadedSignal is raised. See: Exception Handling for a comprehensive list of exceptions your interface code may raise.

Libraries and Environment Variables

VisualWorks provides two useful conventions for identifying composite file paths that may be used across a number of platforms. You can make use of these conventions when designing your ExternalInterface classes for different platforms.

The first convention enables you to specify attributes for libraryFiles or libraryDirectories that include square brackets, such that the bracketed string is taken to be a pattern which is matched against the current platform ID. If it matches, then the library will be loaded. For example, if libraryDirectories included the following names [alpha_osf]dllcc.adux [solaris]dllcc.solaris [win32*i386]dllcc.nt [win32*AXP]dllcc.ant, then the file dllcc.solaris would only be searched for if the OSHandle's currentPlatformId matched *solaris*. Note that underscore characters are mapped to spaces, so dllcc.adux is searched if the platform ID matches *alpha osf*. For additional details, examine the mapping mechanism in ExternalLibraryHolder>>findFile:inDirectories:.

Note that when an interface uses this mechanism to optionally load libraries based on the platform it must also provide an exception handler for the libraryNotFoundSignal, which is raised when an attempt is made to load a library that doesn't exist (for examples, see: Examples and the DLLANDCTestInterface in the DLL and C Connect Test Suite parcel **DLLCCTestSuite.pcl**.)

The second useful convention enables you to use environment variables that are expanded within both the libraryFiles and libraryDirectories attributes. VisualWorks follows the UNIX convention; that is, an environment variable is some alphanumeric string starting with a dollar character ($) and enclosed in parentheses. Thus, under MS-Windows the following directory name expands to the **system32** subdirectory of the current Windows directory: **$(windir)\system32**.

Using these two conventions together, you can set up the libraryFiles and libraryDirectories attributes of your ExternalInterface classes as follows:

```
libraryFiles: '[unix]libc.so [win]msvcrt20.dll [win]msvcrt40.dll '

libraryDirectories: '[unix]/usr/shlib [unix]/usr/lib
  [win]$(windir)\system [win]$(windir)\system32 '
```

This example would work across a number of different platforms. For example, on platforms whose currentPlatformID matches `*unix*`, the library libc.so will be searched for in /usr/shlib and /usr/lib. On platforms whose currentPlatformID matches `*win*`, the libraries **msvcrt20.dll** and **msvcrt40.dll** will be searched for in **%windir%\system** and **%windir%\system32**.

When using environment variables, be aware that the Object Engine sets these variables only once when you start a virtual image, and that any subsequent changes made to the environment variables in the operating system will not be reflected for mapping the path(s) of your library files. This is a consequence of the way that most operating systems pass environment variables to processes (in this case, the Object Engine).

Programmatic Search

The class definition template provides a fixed means for you to specify the library files and their directories. However, there may be situations where these lists are not known at compile-time, but can only be determined at run-time. In these situations, you can use the following ExternalInterface class protocol to define the library list and the library directory list. Send these messages directly to your interface class.

Table 3: Defining the library list and library directory list

Method name	Purpose
libraryFiles:	Reset my libraryFiles. The libraries are unloaded immediately, so that a new search for the affected libraries will occur the next time they are invoked.
libraryDirectories:	Reset the private part of the library path, which is linked to the superclass's library path. To have this take effect immediately, this method will unload the library and the subclasses' libraries, so that a new search for the affected libraries will occur the next time they are invoked.
unloadLibraries	Unload the libraries without changing how they are specified. The next time the libraries are accessed, it will be relinked.

Parsing C Header Files

The StandardLibInterface example used a feature of DLL and C Connect that enables you to parse C language header files and generate interface methods that correspond to the declarations contained in those files. The parser recognizes the following C language declarations:

- typedefs
- enumeration types and constants
- macros
- defines
- function prototypes
- variables

In addition, the parser recognizes the following standard preprocessing directives:

- #if, #ifdef, #ifndef, #else, #elseif, #endif
- #define
- #pragma
- #line

- #error
- #include
- #assert

The parser evaluates all preprocessor directives. However, it effectively ignores #pragma which contains compiler-specific directives. In addition, the #line directive is simply ignored.

When you define an interface class and specify header file names in the includeFiles class attribute, the class stores the header files that can be subsequently parsed. You may then use the External Interface Builder to parse the specified header files, and then to compile methods into your interface class that correspond to the declarations contained in the header files. For details about using the Builder tool, see the discussion of the External Interface Builder Tool.

You can also control which declarations in the file are automatically generated, by using a special pattern-matching string as the argument to the class-side method generateMethods: in ExternalInterface class. For example, if you only wanted to use the multiply() function in a large library, you could avoid generating all the other methods by providing 'multiply' as the argument to the generateMethods: keyword.

```
MyInterfaceClass generateMethods: 'multiply'.
```

Wildcarding is recognized in the pattern-matching string, and you can include multiple patterns separated by spaces.

The following table summarizes the variations that you can use for the argument to ExternalInterface class>>generateMethods:

Table 4: Wildcard variations

Argument	Result
'' [empty string]	(Re)generate none
' ' [space]	Regenerate existing methods only
'add sub*'	(Re)generate add and any externals beginning with sub, in addition to existing methods

Argument	Result
'*'	(Re)generate methods for all externals declared in the header files

You should be very careful about which header files you parse using this technique. Many header files include other header files, which recursively include yet more header files. It is very easy to specify a system header file that eventually produces an interface class with hundreds or even thousands of interface methods. Each interface method consists of several bytes of information, potentially adding thousands of bytes of method definitions to your VisualWorks image. Be very careful to create and parse header files that contain the bare minimum of interface declarations. Make sure you remove automatically generated method declarations that are not required by your external interface class.

When you generate an interface by parsing a header file, the interface class automatically defines the following selector protocol. You can create your own protocol and move the methods to a more descriptive location if you want to change the default naming.

Table 5: Selector protocol contents

Protocol	Contents
macros	All the macros in the header files. Macro is defined as a #define that accepts arguments.
defines	All the #define statements in the header files.
types	All the typedef statements in the header files.
variables	All the global variables in the header files.
procedures	All the function prototypes in the header files

To clarify this mechanism, we can return to the example class StandardLibInterface, which was described earlier in this chapter.
To parse the C header file that describes this interface, the class generator needs to search for and read the contents of the **`stdlib.h`** file. If you have not specified a full path name to the file, or have not specified any directories in the includeDirectories attribute, the system assumes the header file is located in the current directory.

If a header file cannot be found, a dialog is displayed with an error message indicating this fact. Click the **OK** button to proceed. You should then check the includeFiles and includeDirectories attributes of the interface class definition for invalid entries or check for the existence of the named file to correct the error.

If, for example, you selected the abs() function prototype, an instance method named abs: will be created automatically. When the DLL and C Connect parser reads the file **stdlib.h**, it uses the function prototype to create the abs: method. Your C code has now been registered. The abs() function in the DLL will be linked the first time you invoke it.

The parser evaluates #define (defines and macros) statements that evaluate to scalar values (numbers) and strings. No claims are made that the parser correctly evaluates more complex C code expressions.

For example, the following evaluates correctly because it evaluates to a scalar:

```
#define CONSTANT ((long) sizeof(int) * 4) + ((short) 3 << 8)
```

The following evaluates correctly because it also evaluates to a scalar:

```
#define STRING "string"
```

However, the following #define statement contains code fragments, and does not evaluate to a scalar:

```
#define MEMBER_ACCESS(a) ((a) -> foo)
```

Pre-Defined Constants

The C declaration parser defines the following predefined macro names, as specified by the ANSI C standard.

Table 6: ANSI C predefined macro names

__LINE__	Line number of the current source program line
__FILE__	Name of the current source file

__DATE__	Calendar date of the translation
__TIME__	Time of the translation
__STDC__	Set to 1 if the compiler is ANSI C

If your C compiler is nonstandard, then some or all of these declarations may be missing. Your C compiler may also supply other predefined macros for your use.

Currently, `__DATE__` and `__TIME__` work as specified in the ANSI C standard. `__STDC__` indicates where ANSI C (as opposed to K&R) is being parsed. If you want your C code to be able to be compiled by both an ANSI C or K&R C compiler, then your C code should contain versions for both ANSI C and K&R C wherever the syntax differs.

The different versions can be marked with the following C preprocessor code.

```
#if __STDC__     /* ANSI C code */
#else     /* K&R code */
#endif
```

DLL and C Connect currently does not provide a mechanism to pre-define constants prior to starting a header file parse. This is contrary to most C pre-processors, where a compiler switch is available for such a purpose.

However, you can take advantage of the header file parsing order when defining your `ExternalInterface` subclass. Simply place all predefined constants in a header file and list that header file first in the `includeFiles` attribute of your class definition template. All the `#define` statements in that file will be pre-processed before the remaining header files.

For example, suppose you have a header file designed for use on multiple platforms and wish to create a pre-defined constant indicating on which platform the header file is to be parsed. Assume your library is called `lib.dll` and the library's header file is called `lib.h`. You want to parse the header file for the Win386 platform (this platform identifier is completely arbitrary and is used only for example purposes). First create a header file called `win386.h` that contains the following line (or any other #define statements that you wish to declare):

```
#define PLATFORM platWin386
```

Your header file may then contain conditional compilation directives such as the following:

```
#if PLATFORM == platWin386
 .
 .
 .
#endif
```

To actually build this example interface to the Win386 platform, you would use the following class creation template:

```
Smalltalk defineClass: #Win386
 superclass: #{External.ExternalInterface}
 indexedType: #none
 private: false
 instanceVariableNames: ''
 classInstanceVariableNames: ''
 imports: '
   private Smalltalk.Win386ExternalDictionary.*
   '
 category: 'ExternalInterface-Examples'
 attributes: #(
   #(#includeFiles #('win386.h lib.h'))
   #(#includeDirectories #())
   #(#libraryFiles #('lib.dll'))
   #(#libraryDirectories #())
   #(#beVirtual #false)
   #(#optimizationLevel #full))
```

Again, notice that the pre-definition file, **win386.h**, appears before the library header file.

Syntax Errors

On rare occasions you might attempt to parse a header file that contains constructs not recognized by the C declaration parser. When the parser encounters such an error, it displays a Syntax Error notifier window. The window contains the pre-processed text of the entire header file contents. This includes the top-level header file, and recursively any header files it includes. Because you are viewing pre-processed source code, the actual text does not look the same as the original source.

Within the window a highlighted string, `syntax error ->`, marks the suspected position of the syntax error. This highlighted text is not always visible in the new window. You may need to scroll through the text to the location of the syntax error marker. Although the parser attempts to be as accurate as possible when indicating the location of the error, many times it indicates only an approximate location. You should look several characters or statements before the marker to locate the actual cause of the error.

To proceed from a syntax error, and ignore the current error, you can allow the parser to attempt a recovery by simply selecting `proceed` from the <Operate> menu. Most of the time, however, this results in further notifiers, because the parser is now in a confused state.

To proceed from a syntax error and halt parsing, select `close` from the <Window> menu, or use the window's close mechanism. Once the window is closed, your interface class does not contain any method declarations for the current parse. Edit the header file to fix the syntax error and re-parse the file.

Due to the restrictions on the parsing of `#define` statements, it is possible to encounter trouble with C definitions such as the following:

```
#define APIENTRY _System
[...]
typedef int ( APIENTRY _PFN ) ();
typedef _PFN *PFN;
```

This code will produce a syntax error during parsing. This limitation will be eliminated in a future release of DLL and C Connect. Under the current implementation, you can work around the limitation by creating a "wrapper" header file which `#defines` the offending types to nothing. Since the types are not handled by the DLL and C Connect parser, and are therefore meaningless for Smalltalk, this approach is easier than editing your header files. Including these “wrapper” header files will allow you to parse the files without error.

For example, the previous example can be parsed correctly if it is preceded by a file containing the following wrapper definitions:

```
#define _System
.
(define any other problematic types as well)
.
.
```

When you simply want to see the definitions in a header file, or test to verify that it parsed correctly, you can do so without having to create a new subclass of `ExternalInterface`. A utility protocol provided by class `ExternalInterface` parses one or more files and returns a dictionary containing the names that are defined in the `#include` files. For example, on a Windows NT platform you could inspect the following expression to see the contents of **`stdlib.h`**:

```
ExternalInterface
 parseIncludeFiles: 'stdlib.h'
 includeDirectories: 'C:\MSDEV\include\'
```

When defining a new interface class, you should bear in mind that the superior strategy is to use the External Interface Builder tool (for details and a discussion of its use, see: External Interface Builder Tool.

Interfacing with Other Languages

Instead of calling external code compiled with a C compiler, you might want to call code compiled with a different language compiler. DLL and C Connect can provide access to other languages, as long as the interface to the external code conforms to a calling convention for supported C functions. Be aware that interfaces to external code written in languages other than C are not officially supported.

To access DLLs written in another language, you must use the standard development tools to create an appropriate `ExternalInterface` and associated accessing methods. You can use either the

VisualWorks programming tools, or the special tools that are part of DLL and C Connect.

An example of this feature is to use DLL and C Connect to access dynamic-link libraries built with a C++ compiler. Although calling C ++ member functions or accessing C++ objects are not supported, it is possible to write a C language wrapper to your C++ code.

To do so, you must wrap external library function prototypes within an external C block. Each function declared in this block will conform to C calling conventions. The body of the function can then call C++ member functions and access C++ objects.

For example, you can access C++ code using C wrappers in the following manner:

```
extern "C" {
 int wrapperFunction1(void);
 long wrapperFunction2(long);
}
int
wrapperFunction1(void)
{
 <<C++ code>>
}
long
wrapperFunction2(long aLong)
{
 <<C++ code that uses the argument aLong>>
}
```

The above C++ example does not discuss the problems associated with global variable constructor execution when dynamic-link libraries are loaded. We leave this detail to the particular compiler implementation.

Other language bindings, for example COBOL, may be possible given a C language wrapper layer.

Chapter

3

Creating and Accessing C Data

Topics

A function in an existing C library sometimes requires an argument having a custom data structure, such as a structure containing several members. To create a suitable C data object for such a function, you must be able to create an equivalent object in Smalltalk. In addition, C data objects that survive across function calls or callbacks need to be allocated in a special memory heap where they are protected from the Smalltalk garbage collector.

Memory Allocation in Smalltalk

Smalltalk provides fully automatic memory management that frees developers from the traditional concerns of storage allocation imposed by C. Since the two languages have very different underlying assumptions about memory management, it is absolutely critical that you use the correct strategy for passing parameter data between Smalltalk and C.

To choose the proper strategy for passing data objects between Smalltalk and C, it is important to first look briefly at the way that the Smalltalk Object Engine handles storage allocation.

At system startup time, Smalltalk allocates and controls the following fixed-size memory spaces: CompiledCodeCache, StackSpace, NewSpace, LargeSpace, PermSpace, and FixedSpace. Each of these spaces is used by Smalltalk to hold program elements of a particular type. The default size of most of these spaces can be altered at system startup (see the protocol for ObjectMemory class). For Smalltalk applications that pass data to C functions, the most important of these spaces is FixedSpace. In addition to the above fixed-size memory spaces, the system also manages a variable-size space known as OldSpace.

The CompiledCodeCache contains methods whose byte codes have been compiled directly into machine code for the current platform.

The StackSpace contains the stack of activation contexts created during the execution of a Smalltalk process.

NewSpace is used to house newly created objects. It is composed of three partitions: an object-creation space, which is called Eden, and two survivor subspaces.

LargeSpace is used to hold the data of large byte objects (bitmaps, strings, byte arrays, uninterpreted bytes, and so on). In this case, “large” means byte objects greater than 1 kilobyte.

PermSpace is used to hold all semipermanent objects. Because they are rarely ready to die, the objects held in PermSpace are exempt from being collected by any reclamation facilities other than the global garbage collector.

`FixedSpace` is used to hold semipermanent, non-pointer objects that are passed as arguments to selected C function calls. This space is designed specifically for use with DLL and C Connect. The object bodies held in `FixedSpace` are never relocated, but they are still garbage collected. Objects are promoted to `FixedSpace` automatically when they are passed as arguments to a threaded call (these are discussed in the chapter Threaded Interconnect.

`OldSpace` holds all objects that are not held in one of the fixed-size spaces previously described. Although `OldSpace` can be thought of as a single contiguous chunk of memory, it is implemented as a linked list of segments which occupy the upper portion of the system heap.

Reclaiming Space

Smalltalk has automatic collection and compaction facilities for reclaiming space occupied by objects that are no longer accessible from the system roots. These facilities include: a generation scavenger, an incremental garbage collector, a compacting garbage collector, a global garbage collector, and a data compactor. Smalltalk uses these collection and compaction facilities to provide transparent object allocation, object movement, and object reclamation.

The interaction between the memory management facility and your Smalltalk program is typically transparent. Smalltalk takes care of updating pointers when objects are moved in memory (for example, during a scavenge or compaction).

However, this automatic memory management scheme does not mesh easily with C code. An external C function can be passed references (pointers) to objects, and it may expect to maintain these references long after Smalltalk has had a chance to reclaim memory and potentially move memory objects.

DLL and C Connect provides a means to reconcile all of the differences between the automatic storage management employed by Smalltalk and the manual allocation of C. However, it is critical that you choose the correct strategy for allocating C data objects passed to external functions.

The following section provides a more detailed discussion of allocation strategies. If your application is multi-threaded, also see Managing Data Objects with Multiple I/O Threads.

Allocating C Data Types

In general, your application must ensure that all C data objects remain in a fixed location for a predictable amount of time. Data objects passed to external functions must not be referenced by the external library after they have been deallocated by Smalltalk.

To safely share data objects with external libraries, DLL and C Connect provides two general strategies for creating C data objects that will never be moved by the Smalltalk memory management facilities.

1. The first strategy involves explicitly requesting that an object be created in the external heap. This heap is a segment of memory not controlled by the Smalltalk memory manager, and so objects allocated there will never be relocated. You are responsible for allocating and freeing objects located in this heap.
2. The second strategy involves creating objects in `FixedSpace`, a special zone in the object memory that is not subject to compaction. Objects allocated in `FixedSpace` (like those in the external heap) will never be relocated. However, unlike objects in the external heap, those in `FixedSpace` will be subject to automatic garbage collection.

The advantages and tradeoffs between these two strategies are discussed in a separate section on `FixedSpace` later in this chapter (see Allocating Objects in FixedSpace). It is important to bear in mind that regardless of which strategy you choose, both `FixedSpace` and the external heap are limited resources, so you must be careful to manage them accordingly.

The following diagram summarizes the structure of the Smalltalk `ObjectMemory` in relation to the external heap:

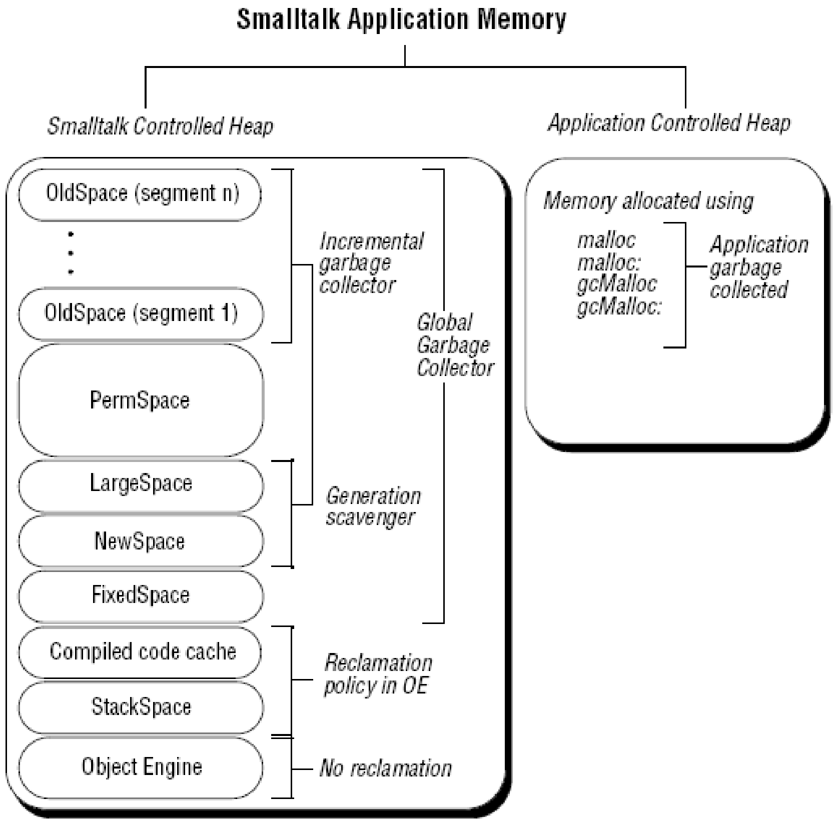


Figure 9: Application memory structure

DLL and C Connect provides six methods that allocate C data types. These methods are `new`, `newInFixedSpace:`, `malloc`, `malloc:`, `gcMalloc`, and `gcMalloc:`. The first two methods, `new` and `newInFixedSpace:`, allocate data in parts of the Smalltalk object memory, so it can be managed automatically by the Smalltalk memory manager.

The remaining methods (`malloc`, `malloc:`, `gcMalloc`, and `gcMalloc:`) allocate data in the external heap. Data allocated in the external heap will not be moved by the Smalltalk memory manager. Thus, references to the external heap can be passed to C functions that store the reference for an indeterminate amount of time, or the references can be passed to functions which invoke callbacks that allow the memory manager a chance to relocate Smalltalk objects. Much like the data in the external heap, objects allocated in `FixedSpace` will not be moved and can be safely manipulated by C functions.

The following table is an overview of the C object allocation protocol that you can send to the objects answered by the `typedef` methods in your interface class.

Table 7: C object allocation protocol

Method	Description
`new`	Allocates a C object in Smalltalk memory. This object will be under the control of the Smalltalk memory manager, and thus will be moved in memory, and its space reclaimed when no Smalltalk object continues to reference it.
`newInFixedSpace: aSize`	Allocates a C object in Smalltalk memory. This object will be under the control of the Smalltalk memory manager, but it will never be moved in memory. Its space will be reclaimed when no Smalltalk object continues to reference it.
`malloc`	Allocates a C object in the external heap. Enough memory is allocated to contain one copy of the receiver. This memory is not controlled by the Smalltalk memory manager and it must be explicitly released by sending the free message to the object answered by the malloc method.
`malloc: numCopies`	Allocates a C object in the external heap. Enough memory to contain numCopies of the receiver is allocated. A pointer to the first element is answered. This memory is not controlled by the Smalltalk memory manager and must be explicitly released by sending the free message to the object answered by malloc:.
`gcMalloc`	Allocates a C object in the external heap. Enough memory is allocated to contain one copy of the receiver. This memory is not controlled by the Smalltalk memory manager. The memory will be automatically released when no Smalltalk object references the object answered by this method. Do not send free to the object answered by gcMalloc — this is handled automatically.
`gcMalloc: numCopies`	Allocates a C object in the external heap. Enough memory is allocated to contain numCopies of the receiver. A pointer to the first element is answered. This memory is not controlled by the Smalltalk memory manager. The memory will be automatically released when no

Method	Description
	Smalltalk object references the object answered by the method. Do not send free to the object answered by gcMalloc: — this is handled automatically.
free	Release the memory allocated by the malloc and malloc: methods. Do not send free to an object referencing memory that was allocated by gcMalloc or gcMalloc:.

Note: The Solaris platform will reserve all memory allocated on the C heap for use by the virtual machine, and will not return this reserved memory to the operating system until the virtual machine process terminates. This can cause the memory footprint of the virtual machine to grow significantly, in particular when using the gcMalloc: mechanism. Depending on the system configuration, this may lead to an inability to execute external processes due to lack of enough virtual memory to execute fork() successfully. See the manual pages for fork() and free() in your Solaris installation for more information.

If the memory manager is unable to meet an external heap allocation request to malloc, malloc:, gcMalloc or gcMalloc:, an exception will be raised with the error #'C allocation failed'. The parameter field in the SystemError object associated with the exception may contain the number of bytes needed to satisfy the allocation request. Your code should include an exception handler to catch these errors. For more details on the specifics of error handling, you may refer to the section Exception Handling.

The allocation strategy that you choose depends upon several factors. For data that you would like to persist in the form of Smalltalk objects (for example, across snapshots), it may be preferable to use FixedSpace. Tradeoffs in this strategy include the requirement that pointers must be updated after snapshots, and that only byte-field objects may be allocated in FixedSpace. Objects allocated by calling newInFixedSpace: can be more efficiently manipulated by both Smalltalk and C, but they also involve additional storage overhead. For data that you would like to persist in the form of C data objects (e.g., across several C function calls), the best approach may be to use malloc or gcMalloc.

Allocating space on the external heap for a C data object is performed with a `malloc` message. This message is sent to an instance of subclass `CType`, which represents the type of data to be allocated.

To illustrate the use of `malloc`, consider the following hypothetical example class `CustomerInterface`, which will be used to instantiate a series of `Customer` objects. The memory requirements of the instances are described by class `CustomerType`, which is a subclass of `CType`. Assume that these classes are generated during the automatic parsing of a C interface specification. To allocate space on the external heap for the `CustomerType` object, you would use the following expression:

```
pointer := CustomerInterface new CustomerType malloc.
```

You can also use the `malloc:` message to allocate multiple `CustomerType` slots on the heap:

```
pointer := CustomerInterface new CustomerType malloc: 5.
```

The result is a `CustomerType *` pointer, which points to the first element of an array for five `CustomerType` C data objects.

To deallocate the heap space associated with a pointer, send `free` to it, as in the following example:

```
pointer := CustomerInterface new CustomerType malloc.
"pass the pointer to functions that use its datum."
pointer free.
```

You can also use a variant of `malloc` (`gcMalloc` or `gcMalloc:`) to allocate garbage-collectable space on the heap. When the pointer to storage allocated using `gcMalloc` is no longer referenced within Smalltalk, the space on the heap is freed automatically. An example is:

```
| pointer |
pointer := CustomerInterface new CustomerType gcMalloc.
"Pass the pointer to functions that use its datum"
```

When the above code is completed, and the pointer variable no longer exists, the `CustomerType` pointer object is no longer referenced by any Smalltalk object, so it will be deallocated the next time the

Smalltalk garbage collector finalizes objects. Note that you do not need to be concerned with the finalization process here. For more information on finalization, consult the *Application Developer's Guide*.

C procedures that retain copies of pointer arguments can cause referencing problems. If you use gcMalloc, gcMalloc:, or newInFixedSpace: to allocate external memory subject to automatic garbage collection and pass the associated CPointer to external code that maintains a copy of this pointer, you must maintain a Smalltalk reference to that CPointer for as long as the external copy of the pointer is valid. If the CPointer is dropped and garbage collected, the block of referenced memory is freed, and may be reallocated. When the external C code attempts to use its (now stale) copy of the pointer, it may find garbage where you once had valid data. This can be a very difficult problem to isolate, and is potentially fatal. Therefore, take care that your CPointer objects are referenced for the lifetime of their corresponding heap pointer.

Allocating Space on an External Heap

Of the two allocation strategies described at the beginning of this chapter, the first strategy is used to allocate new C data objects on the external heap. This allocation protocol is comprised of the methods malloc, malloc:, gcMalloc, and gcMalloc:. A variant of this strategy involves those situations where you may want to begin with a Smalltalk object and then move it onto the external heap. This is typically done in cases where the C code references the object indefinitely. To copy a selected set of Smalltalk objects to the heap, use one of the following methods:

Table 8: Copying Smalltalk objects to a heap

Method	Description
copyToHeap	Copies the receiver to the external heap and answers a pointer to the heap location. This memory is not controlled by the Smalltalk memory manager and must be explicitly deallocated by sending the free message to the answered object.
gcCopyToHeap	Copy the receiver to the external heap and answer a pointer to the heap location. This memory is automatically deallocated

Method	Description
	when the Smalltalk object answered by the message is no longer referenced.

The heap copy protocol cannot copy an arbitrary Smalltalk object (or object graph) to the heap. It is designed to copy scalar objects (numbers, strings, and byte arrays). The classes whose instances can be copied directly with this protocol are as follows: Integer, Double, Float, Character, ByteArray, ByteEncodedString (and its subclasses), IntegerArray, TwoByteString, UninterpretedBytes, and WordArray.

To copy an instance of a Smalltalk object to the heap, send it a copyToHeap message, as follows:

```
pointer := anInteger copyToHeap.
```

Since external heap space that is allocated in this manner is not subject to automatic storage management, your application must explicitly deallocate this memory with a free message. An alternative approach is to use the message gcCopyToHeap to return a pointer that is marked as being collectible. When the Smalltalk pointer object is no longer referenced, its associated heap space is freed automatically.

To copy a Smalltalk object to the heap and return a pointer that is subject to garbage collection, use an expression such as the following:

```
pointer := anInteger gcCopyToHeap.
```

The following table indicates which type of C pointer is returned when copying a Smalltalk object of a particular class. With respect to the abstract class ByteEncodedString, it should be noted that the pointer attribute also pertains to each of its subclasses, and not simply the abstract class as the table might seem to suggest. (This includes ByteString, MacString, OS2String, and ISO8859L1String.)

Table 9: Pointer returned after a copy

Class	Pointer
Integer	long *

Class	Pointer
Double	double *
Float	float *
Character	unsigned char * *or* unsigned short *
ByteEncodedString	unsigned char *
TwoByteString	unsigned short *
ByteArray	unsigned char *
WordArray	unsigned short *
UninterpretedBytes	unsigned char *

You must override the implementation of copyToHeap if you want to write specialized code to copy instances of one of your application classes to the heap. It is impossible to copy an arbitrary Smalltalk object to the heap, as the object can be a complex graph with no direct C representation.

For example, suppose your application contains a class that references a Smalltalk array of float objects. You determine that when this class is copied to the heap, the application's behavior that you defined requires that the class creates a C float array containing the information in the Smalltalk float array. To do this, you can use the following code fragment to perform the copy. Assume that the class instance variable that references the Smalltalk Array object is called floatArrayInstVar.

```
copyToHeap
 | floatArrayPtr arraySize tempPtr |
 arraySize := floatArrayInstVar size.
 floatArrayPtr := CLimitedPrecisionRealType float malloc: arraySize.
 tempPtr := floatArrayPtr copy.
 1 to: arraySize do: [:i |
  tempPtr contents: (floatArrayInstVar at: i).
  tempPtr += 1].
 ^floatArrayPtr
```

Note the following about the above code fragment:

- The C float type object allocated enough storage in the external heap to contain all the array's Float data. It constructs a C float type

(CIntegerType float) and then calls malloc: with the correct size of the array as a parameter.

- A temporary pointer was created by copying the heap float array pointer (floatArrayPtr). This is required because the copy loop destructively modifies the loop pointer using the += message. You must maintain a pointer at the beginning of the array, so it can be answered as the value of copyToHeap.

Creating C Data

All generic C data types are available for use by your Smalltalk code. These data types are defined by special interface classes that are part of DLL and C Connect. This includes scalar types (integer, float, character, and enumerated), composite types (union, structure, and array), and pointer types.

DLL and C Connect provides a class hierarchy that parallels C data types. CDatum and its subclasses represent actual C data, such as string, or the pointer represented by an address, such as 0x734521.

CDatum subclasses provide the means to represent more complex data types, such as structures and arrays. When a C function has an argument or return value of one of these data types, DLL and C Connect converts between the C data and an instance of a subclass of CDatum. The subclasses of CDatum also provide appropriate accessing methods so that your application code can insert data into and extract data from the CDatum.

Each CDatum contains an object to hold the actual data, plus an instance of CType to represent the C type of the data.

For simplicity, you should avoid creating instances of CType and CDatum objects by directly sending allocation messages to these classes. Instead, create them as a side-effect of access methods defined in their ExternalInterface subclass and the type instance creation messages (new, newInFixedSpace:, malloc, malloc:, gcMalloc, and gcMalloc:.)

Scalar Data

Scalar value types are represented directly by ordinary Smalltalk objects: Integer, Double, and Float. For example, a function that expects an integer as an argument can be passed an instance of SmallInteger, LargePositiveInteger, or LargeNegativeInteger. Conversion to and from the correct integer class is handled automatically. A #'bad argument' exception is raised when an incorrect value is passed, such as when a LargePositiveInteger is passed to a function that expects a short integer (for details on exceptions, see Exception Handling.

The following table lists the corresponding scalar types:

Table 10: Correspondence between scalar C types and Smalltalk classes

C type	Smalltalk object
unsigned char	SmallInteger
char	SmallInteger
unsigned short	SmallInteger
short	SmallInteger
unsigned int	SmallInteger or LargePositiveInteger depending on a particular platform's int type size.
int	SmallInteger, LargePositiveInteger or LargeNegativeInteger depending on a particular platform's int type size.
unsigned long	SmallInteger or LargePositiveInteger
long	SmallInteger, LargePositiveInteger or LargeNegativeInteger
float	Float or Double
double	Float or Double. If Float is used, it is promoted to Double-precision.

Notice that SmallInteger, LargePositiveInteger, and LargeNegativeInteger are used to represent [unsigned] int and [unsigned] long data types. Smalltalk imposes a range limit on SmallIntegers, typically -229 to 229 - 1. Depending on the platform's default byte size for these types, the maximum or minimum SmallInteger values can be exceeded. In this case Smalltalk uses a LargePositiveInteger or LargeNegativeInteger representation. However, not all LargePositiveInteger or LargeNegativeInteger objects can be used, only those that fall into the default range of the platform's byte size for [unsigned] int and [unsigned] long types. If an out-of-range Smalltalk number is used for a given C scalar type, your C function call will fail with a #'bad argument' exception.

For float and double scalars, use both Smalltalk Float and Double objects. Note, however, that every Float can be coerced into a Double, but not every Double can be coerced into a Float. If you use a Double when a float is expected, your C function call will fail with a #'bad argument' exception that identifies the invalid argument (for details on exceptions, see Exception Handling.

A Smalltalk Character object does not undergo a conversion process. You are responsible for converting the character object to and from an integer value using the current platform's encoding. You can convert the character by performing the following:

```
anInteger := String defaultPlatformClass encode: aCharacter
aCharacter := String defaultPlatformClass decode: anInteger
```

Do not assume the asInteger message returns the correct Character encoding.

Enumeration Types

DLL and C Connect represents enums (enumeration types) using instances of CEnumerationType. For example, an interface class might include the following declarative method:

```
fewMonths
 <C: enum months{ Jan, Feb, Mar, Oct = 10}>
```

You can access members of this enum using an instance method e.g.:

```
getMonth
 | anEnum |
 anEnum := self new fewMonths.
 anEnum memberNamed: #Feb.
 anEnum memberNamed: #Oct.
```

The last two lines of the method return the small integers 1 and 10 respectively. If the enum definition included a typedef, i.e.:

```
fewMonths
 <C: typedef enum {Jan, Feb, Mar, Oct = 10} months>
```

Then the getMonth method shown above would need to be changed:

```
getMonth
 | anEnum |
 anEnum := self new fewMonths.
 anEnum type memberNamed: #Feb.
 anEnum type memberNamed: #Oct
```

Note that you cannot change the value of the member outside its definition (this is consistent with the semantics of enums in C). Also, you cannot pass an enum object directly to a C function. Instead, you must pass the members as Integers.

Composite Data

Your interface class can declare structures or unions using a typedef method. For example, suppose you want to pass a structure containing customer information to a function. The structure contains two members: the customer's name and account number. The definition appears in a C header file, and the typedef declaration in the header file resembles the following:

```
typedef struct {
 char *name;
 int account;
} Customer;
```

When you use the External Interface Builder to parse the header file and compile the interface class, a Smalltalk method named Customer is automatically created in the interface class that parallels this declaration. You can also manually create this method by simply adding the method to your interface class. The resulting Smalltalk method would look like the following:

```
Customer
 <C: typedef struct {
  char *name;
  int account;
 } Customer>
```

To create an empty instance of a customer structure, use the following code fragment:

```
| interface customerType customerRecord |
interface := CustomerInterface new.
customerType := interface Customer.
customerRecord := customerType gcMalloc.
```

Notice the following about the above code fragment:

- The second line creates an instance of a hypothetical interface class, the class that encapsulates the `Customer` interface and defines the `Customer typedef` method.
- The third line asks the interface for the `Customer` type. The type is an object that represents the C `typedef` declaration.
- The fourth line asks the type object to create a new instance of itself. The `gcMalloc` message is used to allocate the object in the external heap, so that it will be suitable to pass as a parameter to C code. Note that once the code fragment is completed, the `customerRecord` will no longer be referenced by any Smalltalk object, and will thus be reclaimed during the next finalization.

The expressions for creating an instance of a customer structure can be further condensed, as shown in the first expression below. The second expression places an actual number into the `account` member, using a `memberAt:put:` message, and retrieves the account field using `memberAt:`.

```
| customerRecord |
customerRecord := CustomerInterface new Customer new.
customerRecord
 memberAt: #account
 put: 346.
^customerRecord memberAt: #account
```

The above method simply returns the number 346, the account member.

There may be situations where you need to access a structure member that is a structure itself. The process to access the structure member depends on the method in which the structures were created. The following strategy should be employed when the structure is allocated in the external heap, using the `CType` protocol `malloc`, `malloc:`, `gcMalloc`, or `gcMalloc:`. The example structure is:

```
subStruct
 <C: typedef struct {
  int A;
  int B;
 } subStruct>

baseStruct
 <C: typedef struct {
  char *name;
  subStruct number;
 } baseStruct>
```

Assume the following C function is defined:

```
printStruct: arg
 <C: int printStruct(baseStruct * )>
```

Data can then enter into the structures with the following code fragment:

```
| aBaseStruct aSubStruct customer name |
customer := CustomerInterface new.
name := 'Cincom' copyToHeap.
aBaseStruct := customer baseStruct gcMalloc.
aBaseStruct memberAt: #name put: name.
aSubStruct := aBaseStruct refMemberAt: #number.
aSubStruct memberAt: #A put: 16.
aSubStruct memberAt: #B put: 20.
(customer printStruct: aBaseStruct) inspect
name free.
```

However, when the structure is allocated in Smalltalk memory using the `CType` protocol `new`, use the following method. After `aSubStruct` is populated with data, it should be placed into `aBaseStruct`:

```
| aBaseStruct aSubStruct customer name |
customer := CustomerInterface new.
name := 'Cincom' copyToHeap.
aBaseStruct := customer baseStruct new.
aBaseStruct memberAt: #name put: name.
```

```
aSubStruct := customer subStruct new.
aSubStruct memberAt: #A put: 16.
aSubStruct memberAt: #B put: 20.
aBaseStruct memberAt: #number put: aSubStruct.
(customer printStruct: aBaseStruct gcCopyToHeap) inspect.
name free
```

Those accustomed to writing C code (as opposed to Smalltalk code) might notice a design philosophy that differs slightly from the C philosophy regarding composite member accessing. DLL and C Connect implements a simple load/store/reference design for composite members. All member access protocols either answer a copy of a member (load), sets the value of a member (store), or answer a reference to a particular member. This is contrary to the C language notion of an `lvalue` and `rvalue` (consult a C language reference manual if you are not familiar with this terminology), where the same C expression is used to both load and store values into a member (or expression in general). Keep this in mind when writing Smalltalk code that attempts to mimic existing C code.

In general, the Smalltalk composite accessing protocol answers a copy of a member that is independent of the member's type. The modification protocol sets the value of a member only if the protocol's argument is of the same type as the member. This is true even if a structure member is itself a more complicated type, such as another structure, union, or array. For example, assume the following structure declaration:

```
struct {
 <T1> member1;
 <T2> member2;
 <T3> member3;
}
```

The protocol `memberAt:` answers a copy of `member1`, `member2` or `member3` that is independent of the types `T1`, `T2`, or `T3`. Similarly, `memberAt:put:` fails if the second argument is not an object type equal to `T1`, `T2`, or `T3`. If you do not wish to receive a copy of a member (for example, you want to modify the member in place) use the `refMemberAt:` message.

An exception to this rule is array member accessing. If a structure contains a member that is an array of objects, accessing that member will answer a pointer to the first element of the array (zero-based). For example, assume the following structure definition:

```
PersonName
 <C: typedef struct {
  char firstName[10];
  char lastName[40];
 } PersonName>
```

Creating an instance of the structure and asking for the `firstName` member will answer a `char *` pointer object to `firstName[0]`.

```
| aPersonName firstNamePtr |
aPersonName := CustomerInterface new PersonName gcMalloc.
firstNamePtr := aPersonName memberAt: #firstName.
firstNamePtr contents: 97
```

Pointer Data

C pointers may be used in Smalltalk code. You can create an instance of a C pointer object in one of three ways:

- Allocate space on the external heap
- Copy a Smalltalk or C object to the heap
- Receive a pointer as the result of a function call

Using the `Customer` structure from the previous example, assume the following function declaration. This creates a new `Customer` object and answers a pointer to that object:

```
Customer *CreateDefaultCustomer(void);
```

The following interface method corresponds to the above C procedure declaration:

```
CreateDefaultCustomer
 <C: Customer *CreateDefaultCustomer(void)>
```

The following expressions can be used to access the name member of the `Customer` returned by `CreateDefaultCustomer`:

```
| customerPtr |
customerPtr := CustomerInterface new CreateDefaultCustomer.
customerPtr memberAt: #name.
customerPtr free.
```

Note: Use the Smalltalk value `nil` to represent NULL C pointers.

Array Data

It is a well-known feature of the C language that the distinction between array data and pointers is not very strong. Pointers can be used as if they were declared as arrays. If these pointers are used to index array data, bounds checking on the array object is typically not performed.

When using arrays and pointers, it is very important to remember that C array accessing is 0-based as opposed to Smalltalk's 1-based array accessing.

An example array type declaration is as follows:

```
FloatArray
 <C: typedef float FloatArray[10]>
```

To create an instance of this array of 10 floats in the external heap, use the following code:

```
MyInterface new FloatArray malloc.
```

To access an element of the array, use the following code:

```
| aFloat floatArray |
floatArray := MyInterface new FloatArray malloc.
floatArray at: 3 put: 1.234.
aFloat := floatArray at: 3.
floatArray free
^aFloat
```

Multi-dimensional array protocol is not supported. To access an element of a multi-dimensional array, you must calculate the location of the element yourself. For example, assume the following 10x10 float matrix declaration:

```
Matrix
 <C: typedef float Matrix[10][10]>
```

You can access the float located in the third row and the fourth column by using the following:

```
| aMatrix aFloat threeAtFourIndex |
aMatrix := MyInterface new Matrix malloc.
threeAtFourIndex := 3 * 10 + 4.
aMatrix at: threeAtFourIndex put: 1.234.
aFloat := aMatrix at: threeAtFourIndex.
aMatrix free.
^aFloat
```

It is also possible to obtain a pointer to a sub-element within a multidimensional matrix by simply performing pointer arithmetic on the array pointer. The `refAt:` method can be used to index a pointer into an array.

```
| aMatrix aFloat aFloatPtr threeAtFourIndex |
aMatrix := MyInterface new Matrix malloc.
threeAtFourIndex := 3 * 10 + 4.
aFloatPtr := aMatrix refAt: threeAtFourIndex.
"Same as:"
"aFloatPtr := aMatrix + threeAtFourIndex."
aFloatPtr contents: 1.234.
aFloat := aMatrix at: threeAtFourIndex.
aMatrix free.
^aFloat
```

In the fourth line of this example code fragment, the `refAt:` message returns a pointer that is `threeAtFourIndex` elements from the origin of `aMatrix`. The argument to the `refAt:` method must be a zero-based index.

String Data

Smalltalk String objects are treated as a special case when passed as function arguments typed as char *.

In the case of String arguments, a direct pointer to the Smalltalk String is passed if the String can be properly NULL terminated. If not, Smalltalk makes a NULL terminated copy of the string in Smalltalk's object memory and a pointer to the new copy is passed as the function's argument. It should be noted that objects allocated in FixedSpace are always created with extra space for NULL termination, and thus it may be more efficient to allocate String objects using newInFixedSpace:.

Because of Smalltalk's ability to copy the String argument, do not assume that strings can be destructively modified. If you wish to destructively modify a string, first copy the string to the heap and then copy it back into a Smalltalk String object after the function returns, or allocate the character array directly in either FixedSpace or the external heap.

Care must be taken when passing String objects. Smalltalk implements an assortment of string classes, and each class is used to represent a different type of string. For example, ByteEncodedStrings all use character arrays to represent strings where each character is mapped to a single byte, and TwoByteStrings are used to represent character arrays where each character is mapped to two bytes.

Various platforms maintain different rules about which byte value maps to which character object. To facilitate this mapping in a transparent way, Smalltalk maintains subclasses of ByteEncodedString for various platform character encodings. For example, instances of class ISO8859L1String are used for MS-Windows.

Consistency is maintained by using the current platform's string encoding scheme when sending string object references to C. Every string object undergoes an encoding process if it is not already a platform string. This is true of function arguments. The encoding process may require Smalltalk to make a local copy of the string (similar to the NULL termination scheme described earlier), so do not assume that your C code may destructively modify the string argument. This encoding process will also degrade the performance of your function call, so you should pass Smalltalk String objects with care. An alternative approach is to use newInFixedSpace: to allocate String objects that will not need to be NULL terminated by copying the entire string.

Note that it is possible to implement string classes that are not recognized as such by the virtual machine. In those cases, DLLCC will not automatically null terminate the strings you pass as arguments. Because, in general terms, a C string is a pointer to a character, it may not be possible for the virtual machine to correctly detect every situation that requires null terminating a string. In fact, in some cases it may be better to avoid encoding since the virtual machine may not be aware of the string's encoding (UTF8, UTF16, UCS2, etc). In these cases, it may be better to encode a string into a byte array following the conventions expected by the external function to be called, and then invoke the external function passing a pointer to the first byte in the byte array. You can see examples of this technique by examining senders of encodeWide:.

Casting

It is sometimes necessary to convert one C data type into another form. The C programming language uses a special casting syntax to accomplish this. For example, the following line of C code casts a pointer into an unsigned long value:

```
unsigned long ulObject;
char *szString;
ulObject = (unsigned long) szString;
```

Using the facilities of DLL and C Connect, this same casting operation would look like this:

```
| ulObject szString |
szString := 'daniel' gcCopyToHeap.
ulObject := CIntegerType unsignedLong cast: szString.
```

The first line of this example copies the Smalltalk String object to the external heap. The object szString would be the instance of a CPointer. The second line performs two functions. First, it asks the class CIntegerType for the CType object that represents an unsigned long C type. Then, this type is asked to cast the char * pointer represented by szString. The result is a conversion of the string pointer value into a Smalltalk Integer object. It should be noted that the special case of casting a pointer to an integer scalar type can also be accomplished by sending the message referentAddress to a CPointer object.

External Heap Copying

It is sometimes helpful to perform a wholesale copy of data to and from the external heap. Although the C data type protocol copies individual C data types to and from the heap, there are times where you may need to copy arrays of data, or large chunks of uninterpreted byte data where the data size is only known at run-time.

The C pointer protocol that provides the protocol for bulk data copying is implemented as follows:

Table 11: Copying protocol

Method	Purpose
copyAt: offset from: byteLikeObject size: count startingAt: startIndex	Writes countbytes to the receiver's address + offset from byteLikeObject starting at startIndex. The argument byteLikeObject must indeed be a byte-like object. startIndex must be a positive Integer, and startIndex + the number of bytes copied must be less than or equal to the size of byteLikeObject.
copyAt: offset to: byteLikeObject size: count startingAt: startIndex	Reads countbytes to the receiver's address + offset from byteLikeObject starting at startIndex. The argument byteLikeObject must indeed be a byte-like object. startIndex must be a positive Integer, and startIndex + the number of bytes copied must be less than or equal to the size of byteLikeObject.

Suppose you want to create an array of floating point numbers in order for your C code to manipulate the array. After the C function completes execution, you want to move the entire array back into a Smalltalk object, so that you can further manipulate the data, and have the data stored when you save your image.

The following code fragment shows how to allocate the array in the external heap and copy it into a Smalltalk `UninterpretedBytes` object. It answers two identical arrays. The first array is constructed by copying the data to an `UninterpretedBytes` object; the second array copies the same data but uses the pointer-access protocol to fetch each float data object.

Take some time to study the following example carefully. It contains many of the product’s important programming features.

```
| floatArrayPtr numberOfFloats floatByteSize floatBytes arrayByteSize
 floatArray1 floatArray2 tempPtr |
numberOfFloats := 10.
floatByteSize := CLimitedPrecisionRealType float dataSize.
arrayByteSize := floatByteSize * numberOfFloats.
floatArrayPtr := CLimitedPrecisionRealType float malloc: numberOfFloats.
[tempPtr := floatArrayPtr copy.
"Fill in the array with some data."
1 to: numberOfFloats do: [:i | tempPtr contents: i. tempPtr += 1].
"Between the previous line and the next line, pass the floatArrayPtr to a C
routine that will fill in the Float data. For this example, we omit this step
and simply begin extracting the resulting float data."
floatBytes := UninterpretedBytes new: arrayByteSize.
floatArrayPtr
 copyAt: 0
 to: floatBytes
 size: arrayByteSize
 startingAt: 1.
floatArray1 := Array new: numberOfFloats.
1 to: numberOfFloats do: [:i |
 floatArray1
  at:i put: (floatBytes floatAt: i - 1 * floatByteSize + 1)].
floatArray2 := Array new: numberOfFloats.
tempPtr := floatArrayPtr copy.
1 to: numberOfFloats do: [:index |
 floatArray2 at: index put: tempPtr contents. tempPtr += 1]
 valueNowOrOnUnwindDo: [floatArrayPtr free].
^Array with: floatArray1 with: floatArray2
```

External Heap Alignment

DLL and C Connect is designed to provide seamless access to C functions and data on a variety of hardware platforms running various C compilers. Because of this, each platform and compiler combination enforces certain restrictions on how C data objects must be placed in memory. For example, some platforms do not permit byte addressing, while others impose a performance penalty if data is accessed on a byte boundary, as opposed to the machine's word boundary.

In addition to hardware restrictions, a platform's compiler may impose its own restrictions on the organization of data in memory. This organization is most evident with C structures and unions, where the compiler implements a particular alignment algorithm. This alignment algorithm is a mechanism the C compiler uses to position the data members within a structure.

To ensure that DLL and C Connect works predictably with C libraries or compilers that employ non-standard data alignment algorithms, the product provides a means to define alternate alignment algorithms. This mechanism is useful when the memory layout of C data objects must be under the control of the programmer.

To support each platform and compiler combination, a flexible approach is taken to structure and union layout, by implementing a layout algorithm class called CStructureLayout. This class is responsible for implementing various layout and aligning algorithms for C data structures.

CStructureLayout operates in the following way. Very early in the Smalltalk application start-up sequence, the CStructureLayout class receives an installOn: message. This message is sent by the ExternalInterface class in response to ObjectMemory's earlySystemInstallation change request (see External Interfaces and Snapshots for more details on the earlySystemInstallation change request). CStructureLayout uses the argument to installOn: to determine which platform your application is running on and proceeds to realign all the structure and union objects in the system to use the platform's default alignment algorithm for the current platform.

CStructureLayout implements the most common structure layout algorithm for each platform. However, many compilers allow various layout algorithms to be used by setting certain compile time switches. If you need to implement your own layout algorithm, you can do so by performing the following steps:

- Add a new class instance creation message for your new layout algorithm to either the subclass CStructureLayout, or your own subclass. CStructureLayout implements the layout algorithms using a series of alignment blocks. You may consult CStructureLayout's class comment and the existing layout algorithm implementations for details on how to implement your own layout algorithm.
- Alter the method CStructureLayout>>installOn: to recognize the platform and install your new layout algorithm.
- During a response to ObjectMemory's returnFromSnapshot change request, after CStructureLayout has initialized itself, set the default layout algorithm by sending defaultLayout: to CStructureLayout. The argument is an instance of a CStructureLayout that implements your layout algorithm. Then realign all the structures and unions in the system by sending CCompositeType the realign message.

Unfortunately, the returnFromSnapshot change request may arrive too late in your application's start-up sequence. This opens up the possibility that you will use your interface class before it is properly initialized. Use either CStructureLayout's installOn: method, or the earlySystemInstallation change request for this reason.

Existing libraries assume a particular structure layout algorithm. If you implement your own layout algorithm, or change the default, you must make sure that libraries used with this new layout are compatible.

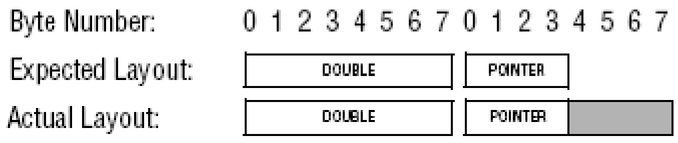
Unexpected Data Alignment in C Structure Objects

To understand how alignment algorithms may be implemented, let's consider the following example.

On the MS-Windows platform, suppose that we have defined the following structure in our ExternalInterface subclass:

```
<C: struct teststruct1 {
 double d;
 char * p;
 }>
```

Ordinarily, we would expect that the compiler aligns the structure elements as follows:

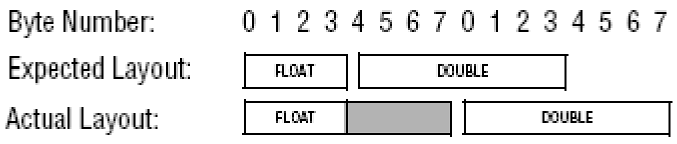


Since the second element in our structure is defined as a pointer, we expect the size of this object to be determined by the size of platform's address-space (in this case, 32-bits). Although we expect that the whole structure only occupies 12 bytes, the compiler adds an extra 4 byte space for alignment (as shown in the lower layout). The pointer object p is still located in the same byte location, but the overall size of the structure is padded with four empty bytes.

A similar situation can be observed in the following struct definition:

```
<C: struct teststruct2 {
 float f;
 double d;
 }>
```

Again, the structure definition produces an unexpected layout, which may be illustrated as follows:



In both of these cases, the compiler sets an alignment and size for the structure elements in order to align them at eight-byte multiples (this is called *natural alignment*, because a double is eight bytes in size). The whole struct object is aligned at eight-byte multiples according to the most-restrictive-member rule; in this case the most restrictive size is double. Moreover, the byte size of the struct must be a multiple of eight even if it contains members with data-sizes less than a machine word in length (i.e., four bytes in the WIN32s environment).

Since the Microsoft WIN32s/NT environment aligns struct objects containing eight-byte data on eight-byte boundaries, DLL and C Connect normally uses this alignment algorithm for proper compatibility. However, some third party libraries may not follow this alignment rule.

Changing the Alignment Algorithm

For compatibility with non-standard libraries, you can implement alternate alignment algorithms. DLL and C Connect provides a special class named `CStructureLayout` to implement this algorithm. By default, this class provides a 32-bit packing algorithm, but by adding new methods, you can support various layout algorithms.

Depending on the alignment you need, it may or may not be necessary to implement a new layout method. Browse the class protocol for `CStructureLayout` to see the standard implementation. provides methods for to DOS layout, two-byte, and four-byte layouts.

You can change the default alignment by evaluating a Smalltalk expression similar to the following (the receiver is a subclass of `ExternalInterface`):

```
SubclassInstance teststruct1
  typeDo: CStructureLayout twoByteLayout
```

Or, alternately:

```
SubclassInstance teststruct1
  typeDo: CStructureLayout dosLayout
```

Note that you must change the default alignment before you allocate a pointer to a structure object. Alternately, you can realign all existing structure and union objects by sending class `CCompositeType` the message `realign`. Also, the alignment will only be applied to the structures associated with the ExternalInterface subclass that receives the `typeDo:` message. If you have multiple interface classes that need to use a different alignment algorithm, you must send this message to each one separately.

For discussions of platform-specific alignment issues, you may consult Platform Specific Information.

External Heap and Snapshots

Smalltalk preserves the semantics of objects across a snapshot in most cases. The procedures to perform this are provided by the snapshot facility itself, which preserves the state of object memory. This is sufficient for objects whose interpretation is completely contained in an image.

However, some types of objects are affected by the external environment and must be given special treatment. In particular, pointers into the external heap that were active when the snapshot was made will be invalid.

For example, suppose you created an object located in the external heap using the `malloc` message sent to a C data type. The C data pointer, represented by an instance of the Smalltalk class `CCompositePointer`, `CProcedurePointer`, or `CPointer`, remains in the image after a snapshot. However, the pointer into the external heap becomes invalid when you restart the image.

On image startup, the external interface machinery clears all your pointer objects so that a subsequent reference will raise an exception rather than dereference an invalid (i.e., `nil`) pointer (for details on exceptions, see Exception Handling. In general, you should try to structure your application to re-construct all external data after start-up.

Allocating Objects in FixedSpace

As an alternative to copying byte arrays or strings between Smalltalk and C data objects, it is possible to allocate the objects in a special zone of the object memory known as `FixedSpace`. Throughout their lifetime, objects allocated in `FixedSpace` are managed by the Smalltalk memory manager but they are not subject to relocation during compaction. When they are no longer referenced by the system, they will be reclaimed by the garbage collector. FixedSpace provides a storage mechanism that satisfies the general storage semantics of both Smalltalk and C.

For efficiency reasons, only object bodies are located in FixedSpace, and only non-pointer objects can have their bodies in FixedSpace. Further, all object bodies in FixedSpace have at least two extra bytes allocated to provide for null-termination of single and double-byte strings.

The Object Engine automatically ensures that the body of any byte-like object that is passed as a pointer argument to a threaded call gets promoted to FixedSpace (threaded calls are discussed in the chapter Threaded Interconnect. This ensures that the garbage collector does not move the object's body during the _threaded call, although the garbage collector might move the object's header (and hence need to change its object pointer). If you are allocating an object in FixedSpace, for efficiency you might want to avoid promoting objects during a call, hence DLL and C Connect provides protocol for instantiating byte-like objects in FixedSpace (described below). By initially allocating the object in FixedSpace, you can avoid the additional overhead incurred when the object body is promoted.

Of the two general strategies for storage allocation that have been described in this chapter, the best approach is often to allocate your objects in FixedSpace using the message newInFixedSpace:. There are a number of advantages to using this approach; first, the data object can be more easily shared between Smalltalk and C, because a C function call isn't required to manipulate the object's fields; second, unlike objects allocated on the external heap using malloc, objects in FixedSpace are automatically garbage-collected; and third, String objects allocated in FixedSpace do not need to be copied to add NULL termination required by C string functions. The decision to use FixedSpace instead of the external heap should include a consideration of the life of the objects that will be shared between Smalltalk and C. For allocating certain types of persistent data, a better approach is to allocate space on the external heap using malloc. In general, if long-term persistence of C data is an issue, it may be better not to use FixedSpace.

The following protocols enable you to instantiate objects in FixedSpace, promote mobile objects to FixedSpace, and to test whether or not objects have been instantiated or promoted to FixedSpace:

Table 12: FixedSpace allocation, conversion, and testing protocol

Method	Description
Behavior >> newInFixedSpace: anInteger	Answer with a new instance of the receiver, a class, with the number of indexable variables specified by the argument anInteger. Arrange that the object's data resides at a fixed address throughout the object's lifetime. Such an object is suitable for passing to foreign code, because it does not move over time, and can be effectively shared between Smalltalk and foreign code. Fail if the class is not bits-indexable, or if the argument is not a positive Integer.
SequenceableCollection >> asFixedArgument	Coerce the receiver to an object whose data resides at a fixed address. If the receiver already has fixed data, return the receiver; otherwise return a copy of the receiver which does have fixed data.
Object >> isFixedArgument	Answer whether the receiver, when passed as an argument through DLL and C Connect, represents data at a fixed address. This is true for objects created via the newInFixedSpace: primitive. Fail if the receiver is immediate, because it has no data.
Object >> hasFixedData	Answer whether the receiver's data resides at a fixed address. This is true for objects created via the newInFixedSpace: primitive. Fail if the receiver is immediate, because it has no data.

For example, to allocate a String object in FixedSpace that is appropriate to your platform, evaluate the following Smalltalk expression:

```
| buffer |
buffer := String defaultPlatformClass newInFixedSpace: 64.
```

The String will be 64 characters long, and contain space for NULL termination.

It is important to bear in mind that if you allocate an object in FixedSpace and pass the associated CPointer to external code that maintains a copy of this pointer, you must maintain a Smalltalk reference to that CPointer object for as long as the external copy of the pointer is valid. If the Smalltalk object in FixedSpace is deferenced, it will be reclaimed and the pointer passed to the external function will become invalid. To avoid serious corruption of the object memory, take care that your CPointers are referenced for the lifetime of their corresponding heap pointer.

Because FixedSpace is not subject to compaction, it is possible that the zone can suffer fragmentation that exhausts the available space. When an allocation request in FixedSpace fails, the object memory will be notified and attempt to grow the space. Severe fragmentation could possibly result in repeated attempts to grow FixedSpace, eventually producing behavior that resembles a storage leak. Any application that makes use of FixedSpace for dynamic allocation should be tested for this potential problem. Be aware that the object memory will coalesce FixedSpace when resuming from a snapshot, so fragmentation problems are only an issue when an image is running for long periods of time.

Since the size of FixedSpace is automatically adjusted by the object memory, there is no public protocol for manually setting the allocation for this zone.

Finally, note that object bodies in FixedSpace are subject to the virtual machine's alignment rules. For 32-bit virtual machines, object bodies are aligned on 4 byte boundaries. For 64-bit virtual machines, object bodies are aligned on 8 byte boundaries. This behavior may pose some problems on platforms such as SPARC, which require special alignment for doubles or 64-bit integers. To ensure that the C counterpart to the Smalltalk image will behave as intended in all cases, you may have to dereference pointers to object bodies using a union. For example:

```
inline double
access_double (double *unaligned_ptr)
{
 union d2i { double d; int i[2]; };

 union d2i *p = (union d2i *) unaligned_ptr;
 union d2i u;
```

```
  u.i[0] = p->i[0];
  u.i[1] = p->i[1];

  return u.d;
}
```

(This code example was taken from: Using the GNU Compiler Collection (GCC).)

Depending on your C compiler, you may or may not have access to compiler switches that cause the compiler to schedule instructions equivalent to the function above. Consult your compiler manual for more information.

Representing C Types

Two class hierarchies are fundamental to the operation of run-time C data accessing in DLL and C Connect. The roots of these two hierarchies are `CDatum` and `CType`. These classes interact in a tight fashion to provide a mapping between Smalltalk data objects and C data objects. Every C data object that cannot be represented directly by a Smalltalk class (such as `Character`, `Float`, `Double`, or `Integer`) must be wrapped by an instance of a `CDatum`. However, in order for a `CDatum` object to know anything about the size or type of the object that it wraps, it must store a description of that object type. It references an instance of `CType` for just this purpose. `CType` and its subclasses represent the various C data types that your C interface may be using. They include C structures, unions, type definitions, enumerations, numeric scalars, and pointers.

The following diagram illustrates the interaction between a `CDatum` instance and its associated type:

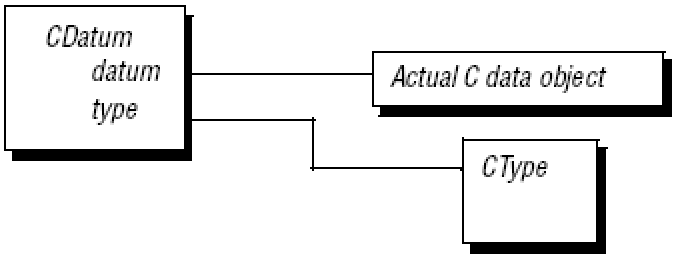


Figure 10: Relationship between CDatum and its associated type

The actual C data object may be located either in the external heap, or in the Smalltalk object memory. As we have already discussed, the allocation protocol for C data objects enables you to choose where the object will be located. Be aware that the manner of representing the C datum will be slightly different in each case.

As an example of the former case (in which the C datum is located in the Smalltalk object memory), suppose your `ExternalInterface` class declared the following C structure type.

```
typedef struct {
 float x;
 float y;
} Point;
```

A method declaration would appear in your hypothetical interface class called `PointInterface`.

```
Point
 <C: typedef struct {
  float x;
  float y;
 } Point>
```

You could create an instance of the `Point` structure using the following expression:

```
PointInterface new Point new
```

Evaluating this expression answers an instance of a `CComposite`, a subclass of `CDatum`, that encapsulates two objects:

- The bytes representing the two float objects contained in the Point structure.
- The type of the object, which in this case is a CCompositeType.

To create an instance of the Point structure on the external heap, you would use the following expression:

```
PointerInterface new Point malloc
```

This expression answers an instance of a CCompositePointer, a subclass of CDatum, that encapsulates two objects:

- An integer value representing the pointer to the bytes in the external heap which contain the two float objects.
- The type of the object, which in this case is a CCompositeType.

In general, instances of class CComposite are used to represent C data objects held in the Smalltalk object memory, while instances of class CCompositePointer are used to hold references to C data objects stored on the external heap. Note that instances of CComposite can be used to hold references to objects on the external heap, but it is the programmer's responsibility to explicitly deallocate these objects when they are no longer in use.

Note: You do not need to create instances of CDatum objects yourself. They are automatically created when allocating CType instances, accessing structure or union members, or calling functions.

Limitations of CType Definitions

Although DLL and C Connect provides a convenient means to define C types using the standard VisualWorks tools, the present strategy used for representing types suffers from a serious limitation. If you are trying to use the Smalltalk inheritance hierarchy to define "abstract" and "concrete" interface classes, you should understand the following limitation.

To clarify the nature of this limitation as well as its possible impact on the design of your interface classes, we shall consider the following two example type definition methods:

```
SomeInterface methodFor: 'types'
```

```
Long
 <C: typedef long Long>

Size
 <C: typedef struct {
   Long cx;
   Long cy;
  } Size>
```

Beginning with the observation that the value of the method `Size` depends upon the method `Long`, we notice that these two methods represent not merely type definitions, but also nodes in a graph of C type objects. In effect, the references from one C type to its subtypes are the arcs in the graph. Here in our example, this arc is the reference from type `Size` to type `Long`.

The complication arises as a side-effect of the way that DLL and C Connect represents these type-relations. When the type definition methods are created, variables which refer to type nodes are stored in each `ExternalInterface`'s type pool. For the example above, `SomeInterface`'s `SomeInterfaceDictionary` would contain `CTypedefType` objects which represent the binding between the name of the type (e.g., `Long`) and the actual `CType` object for every type defined. In this case, the `long` would be represented by a `CIntegerType`. The limitation in this scheme arises because the semantics of a C type dictates that the reference from one C type to its subtypes is direct and does not involve the use of variables (in our example, the contents of the `ExternalInterface`'s type pool).

Because DLL and C Connect uses Smalltalk methods to represent C typedefs, the semantics of the C typing scheme are not reproduced in an entirely faithful manner. As we recall, Smalltalk methods are late-bound via inheritance. This means that in a hierarchy of classes, subclasses may override methods defined in their superclasses, and a message-send will invoke these subclass methods instead of those overridden in the superclass. Since the `typedef` methods used by DLL and C Connect resemble normal Smalltalk methods, it is natural to assume that subclasses may modify the type graph by overriding methods that define C types in superclasses. But since the references between C types are direct, they are not actually affected by the inheritance hierarchy. The immediate consequence of this is that the effect of defining a typedef method in a subclass is not as expected.

The problem is precisely that it is possible to override the root note of a type graph in a subclass. For example, if there were no other typedefs that referred to Size in an ExternalInterface hierarchy that includes SomeInterface, then one of SomeInterface's subclasses could override Size. This is because the Size type refers to other types (e.g., Long), but no other types refer to Size. However, one cannot override an interior or leaf node in the hierarchy. So, for example, if in a subclass of SomeInterface we were to override the Long method, then the Size type would not be affected. This is because the Size type directly refers to the Long type object defined by the Long method in SomeInterface. Reimplementing the method in a subclass does not affect this type graph.

To achieve the desired effect of redefining one type in a subclass, we would have to redefine all the methods in the typing graph. In the given example, this would mean redefining both Size and Long. The effect would be to produce a new type graph independent of the one in SomeInterface, in which the new Size type object directly refers to the new Long type.

The principle consequence of this design limitation is that you must exercise caution when designing any interface that involves a hierarchy of ExternalInterface classes that make use of typedefs. The Smalltalk inheritance mechanism does not allow you to define a complex C type graph by simply overriding selected definitions in your subclass. Note that this limitation does not affect type graphs that you construct using the External Interface Builder tool. You need to be aware of this limitation only when you are using the Smalltalk class hierarchy to define interfaces via inheritance.

Example: Mapping

Assume that a certain C API requires a char* buffer to store a document, and an int to indicate how many bytes are present in the buffer. Moreover, assume the API requires pointers to the char* and int values. Thus, from the DLLCC side, you would need to instantiate a char** and an int*. For the int*, you might use:

```
CIntegerType int malloc: 1
```

or

```
CIntegerType int gcMalloc: 1
```

Both above expressions create a 1-element int array and return the pointer to it. You can access the int value using at: and at:put:.

```
| lengthPointer |
 lengthPointer := CIntegerType int malloc: 1.
 interface doSomethingWith: lengthPointer.
 length := lengthPointer at: 1.
 lengthPointer free
```

For the char**, you will need a pointer to a char*, therefore:

```
bufferPointer := CIntegerType char pointerType malloc: 1.
```

If you wanted to point it to a preexisting buffer, you might want code such as this:

```
| buffer bufferPointer |
 buffer := CIntegerType char gcMalloc: 1024.
 bufferPointer := CIntegerType char pointerType  newOfAddress: buffer
 referentAddress
```

Assuming the API allocates the space for the output buffer and passes it back through the char** parameter, you may need code like this:

```
| lengthPointer bufferPointer |
 lengthPointer := CIntegerType int gcMalloc: 1.
 bufferPointer := CIntegerType char pointerType gcMalloc: 1.
 interface callAPIWith: lengthPointer with: bufferPointer.
 data := ByteString new: (lengthPointer at: 0).
 (bufferPointer at: 0) copyAt: 0 to: data size: (lengthPointer at: 0) startingAt: 1.
```

Protocol for C Data Objects

Subclasses of `CDatum` support a set of messages that mimic C expressions for common operations on C data and pointers. The tables below list these messages, using `cd` to represent C data objects, `cp` to represent C pointers and `ct` to represent C types:

Table 13: C pointer protocol

C pointer expression	Smalltalk expression
cd = *cp	cd := cp contents
*cp = cd	cp contents: cd
*cp1 = *cp2	cp1 contentsFrom: cp2
cp + offset	cp + offset
cp1 - cp2	cp1 - cp2
cp += offset	cp += offset
cp -= offset	cp -= offset
++cp	cp incrementcp += 1
--cp	cp decrementcp -= 1

Table 14: C array protocol

C array expression	Smalltalk expression
cd = cp [index]	cd := cp at: indexcd := (cp + index) contents
cd = cp[index1][index2]	cd := cp at: index1 * rowSize + index2
cp[index] = cd	cp at: index put: cd
cp[index1][index2] = cd	cp at: index1 * rowSize + index2 put: cd
cp1 = &(cp2[index])	cp1 := cp2 + indexcp1 := cp2 refAt: index

Table 15: C structure protocol

C structure expression	Smalltalk expression
cd = cp->member	cd := cp memberAt: #member
cd2 = cd1.member	cd2 := cd1 memberAt: #member
cp->member = cd	cd memberAt: #member put: cd

C structure expression	Smalltalk expression
cd1.member = cd2	cd1 memberAt: #member put: cd2
cp1 = &(cp2->member)	cp1 := cp2 refMemberAt: #member

Table 16: Memory allocation protocol

Memory allocation expression	Smalltalk expression
cp = malloc(sizeof(cdType))	cp := cdType malloc.
cp = malloc(sizeof(cdType) * count)	cp := cdType malloc: count
free(cp)	cp free

The table below outlines the mapping between C data types and their corresponding `CDatum` objects:

Table 17: C data mapping

C datum	Smalltalk datum
<type> *	CPointer
<struct> *	CCompositePointer
<proc> *	CProcedurePointer
<type>[<arraySize>]	CArray
<struct>	CComposite
<union>	CComposite

Chapter

4

Calling Smalltalk From C

Topics

This chapter explores the mechanisms by which DLL and C Connect enables your application to define calls from C code back into Smalltalk.

Defining Callbacks

At times, the relationship between Smalltalk and C should be closer than that provided by a simple call-and-return mechanism. For example, you might want to use Smalltalk's user-interaction facilities to prompt the user for information without exiting from one C function and then calling another to finish the job.

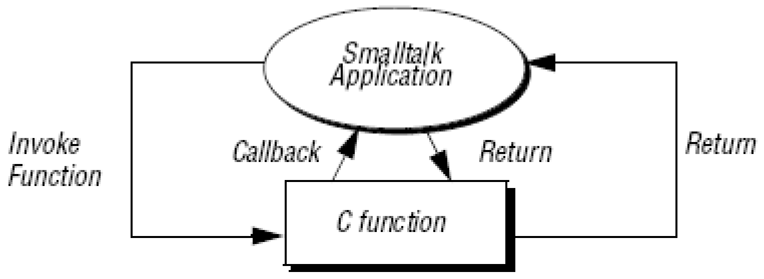


Figure 11: Callback process

As illustrated above, a C function that is invoked from within Smalltalk can also invoke a Smalltalk operation. This temporary exit from C back into Smalltalk is known as a *callback*. DLL and C Connect supports two distinct callback mechanisms:

- An external callback is a Smalltalk block which can be passed to a C function, in order to be invoked as if it were a C function itself.
- An external message-send is a message to a Smalltalk object that can be sent from within a C function.

When considering either mechanism, it is important to remember that callbacks give the Smalltalk memory manager an opportunity to relocate objects. The rest of this chapter demonstrates not only how you can set up callbacks, but also how to arrange for the memory manager to safely update Smalltalk object pointers, which are referenced by the C code.

External Callbacks

In the first type of callback supported by DLL and C Connect, a Smalltalk `BlockClosure` is passed to a C function as a function-pointer parameter. In doing so, the closure can be treated as an ordinary function call. The return value and arguments of the closure must be declared in a typedef statement. The best way to generate the typedef object is to create a typedef method in your interface class.

For example, suppose you have a set of network management functions written in C that you want to invoke from within Smalltalk. One of these functions, called nameNode(), normally calls another C function to prompt the user for the name of a new node on the network. Because the program is running from Smalltalk, substitute a Smalltalk block that prompts the user with a Smalltalk dialog. The following example is a mock-up version of the nameNode() function that is targeted for a UNIX platform. On other platforms, the example may need to be modified (see below):

```
#include <stdio.h>
typedef char *(*GetNameProc)(void);
int nameNode(GetNameProc getName)
{
 char *name;
 name = getName();
 fprintf(stderr, "%s\n", name);
 return 1;
}
```

Next, compile and link this code as a DLL, and then create an interface class.

For the purpose of this example, suppose you are creating a new interface class, called CallbackInterface, whose definition includes nameNode.dll as its library file. You will need to create a type-declaration method (for the block) and a function-calling method for CallbackInterface, as follows:

```
GetNameProc
 <C: typedef char *(*GetNameProc)(void)>

nameNode: aBlock
 <C: short nameNode(GetNameProc aBlock)>
```

So far, you have created:

- A C function nameNode().

- A Smalltalk method (GetNameProc) that declares the function pointer prototype indicating the return type and argument type (in this case, none) of the block to be passed to the C function.
- A Smalltalk method (nameNode:) for invoking the C function and passing a block as the parameter.

In the code that follows, notice that the block is enfolded in an instance of CCallback. The effect of this code is to invoke the nameNode function, which then gives control back to Smalltalk long enough to get the node name from the user. The name is printed by the C function (as a placeholder for the real work that a network management application would do with the name).

```
| externalCallback testInterface returnStatus |
testInterface := CallbackInterface new.
"Create the prompting block in a callback object."
externalCallback := CCallback
 do: [Dialog request: 'Enter a name for the node']
 ofType: testInterface GetNameProc.
"Pass the block to the naming function."
returnStatus := testInterface nameNode: externalCallback.
^returnStatus
```

In the example given above, it was not necessary to guard against the Smalltalk memory manager. The nameNode function does not hold onto any Smalltalk object after the callback is complete. Suppose, however, the nameNode function took two arguments, and the second parameter object was still in use after the callback. In that case, a different strategy would be required: the second parameter object would need to be copied to the external heap from within Smalltalk (as described in the section Allocating C Data Types. By copying the object, it would remain in the external heap untouched even if the Smalltalk garbage collector decided to move things around during the callback. When using this strategy, it is the responsibility of the caller to free the space on the external heap.

An External Callback Example

We can demonstrate the external callback mechanism by using the following simple example based upon a standard C library function. Starting with StandardLibInterface, the example interface class described in the preceding chapters (see Building an Example: StandardLibInterface), we can now extend this class to make use of qsort(), a sorting function in **stdlib**. Since the example class StandardLibInterface already contains the appropriate links to the external library **stdlib**, we need only add the new interface methods.

The C function qsort() uses the QuickSort algorithm to sort an array of integers using a separate function that compares two consecutive elements in the array. The C prototype definition for qsort() is as follows:

```
void  qsort(void*, size_t, size_t, _compare_function)
```

The function takes four arguments: a pointer to the array to sort, the length of the array, the width of the array, and a pointer to the actual sorting function. This sorting function takes two arguments (representing two adjacent elements in the sort array), and returns a result indicating how the elements should be sorted. After repeatedly calling this function, qsort() returns with no result, for the array of elements is sorted in place.

To extend class StandardLibInterface to facilitate calls to qsort(), several new Smalltalk methods are required. The first method defines the function prototype, and would look as follows (note that this method can be generated automatically using the External Interface Builder tool):

```
qsort: arg1 numElements: arg2 ofSize: arg3 with: arg4
 <C: void qsort(
    void * ,
    size_t,
    size_t,
    int (* )(const void * , const void * ))>
 ^self externalAccessFailedWith: _errorCode
```

The second method we need to add to StandardLibInterface defines the typedef for the comparison function called by qsort(). In our example, this function will actually be a Smalltalk block closure. The sorting operation is performed in C, while the criterion for the sorting operation is provided via a block of Smalltalk code. The type definition for this comparison function would be as follows:

```
compareFunction
 <C: typedef int (*compareFunction)(const long * , const long * )>
```

This comparison function takes two long arguments and compares them, returning a negative integer value if the first argument is less than the second, zero if the two arguments match, and a positive value if the first argument is greater than the second.

The interface class must also include a type definition method for the other integer parameters to `qsort()`. This may be defined as follows:

```
size_t
 <C: typedef unsigned int size_t>
```

To actually call the `qsort()` function, we define a last method called `sortExample` to set up all the parameters and then make the call. The following method will sort odd and even numbers into separate groups:

```
sortExample
 "StandardLibInterface new sortExample"
 | sorter type data |
 "define the sorting function."
 sorter := CCallback
    do:
     [:np :mp | | n m |
     n := np contents. m := mp contents.
     n even = m even
      ifTrue: [n - m]
      ifFalse: [n even ifTrue: [-1] ifFalse: [1]]]
    ofType: self compareFunction.
 "create an Array of consequtive numbers that will be sorted."
 type := CIntegerType long.
 data := UninterpretedBytes newInFixedSpace: type dataSize * 50.
 0 to: 49 do: [:i | data longAt: i * 4 + 1 put: i].
 "now, call the sorting function using the sortBlock."
 self qsort: data
   numElements: 50
   ofSize: type dataSize
```

```
  with: sorter.
 ^(0 to: 49) collect: [:i | data longAt: i * 4 + 1]
```

At this point, the code for calling `qsort()` and the code for testing the call are in place. To test this example, you may execute the following Smalltalk expression in a workspace. Select the code and choose **Print It** from the <Operate> menu to see the returned value:

```
StandardLibInterface new sortExample.
```

The result should be:

```
#(0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 1 3 5 7 9
 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49)
```

If you encounter an error while trying to execute this example, make sure that all of the library DLLs are in place (for details, see Building an Example: StandardLibInterface.

Returning From a BlockClosure

It is very important that the return from a closure does not use an up-arrow (`^`) return. Using an up-arrow return can potentially cause stack-overflow problems. This is a limitation of the DLL and C Connect implementation that will be corrected in a future release. The following is an example of a potentially troublesome closure:

```
[:arg1 :arg2 |
 arg1 doSomething
  ifTrue: [^arg2 doTheWrongThing].
 arg2 doSomethingElse.
 ^arg2 doTheLastThing]
```

The return value for the closure should simply be the last statement in the closure. For example:

```
[:arg1 :arg2 | | t |
 arg1 doSomething
  ifTrue: [t := arg2 doTheWrongThing]
  ifFalse:
   [arg2 doSomethingElse.
   t := arg2 doTheLastThing].
t]
```

If the closure uses an up-arrow return, the remainder of the calling C function will not be executed and control will return directly to Smalltalk. A side-effect of this is that the stack of the calling C function will not be cut back. Repeated invocations of the closure will overflow the C-stack.

External Messages

The second type of callback supported by DLL and C Connect is known as an external message-send. This type of callback involves sending a message to a Smalltalk object from a C function. You may execute a series of message-sends during a single function call. Since this approach requires the ability to customize the C code, it is more intrusive than the block-passing mechanism previously described. For example, it cannot be used to provide a callback interface to "off-the-shelf" external libraries.

However, for custom C code which requires a single message-send, external message-sends may be the preferable approach in your interface design. External message-sends are also the only way to pass object pointers (`_oop`) back into Smalltalk from C code.

The Object Engine defines a set of five special message-sending functions. The first C function is for unary messages, and the other four are the one-, two-, three-, and multi-keyword messages. Each function takes arguments that identify the receiver of the message, the message selector, and the arguments (if any). The receiver, message selector and arguments are Smalltalk objects, so in the function templates that follow, an "`o`" (for object) is prefixed to each of the argument variable names.

Another argument to the message-sending function, called `poKeep` (pointer to an `_oop`), is used to store pointers to objects that your C function uses after the callback. It keeps them safe during memory management operations that might occur during the callback. Note, however, that this strategy is only effective for pointers that are held by the first C function called via DLL and C Connect. If a calling function anywhere in the C stack holds a pointer into Smalltalk memory, that pointer could be invalidated during the callback. For this reason, only use callbacks in functions that are invoked directly from Smalltalk, where the poKeep mechanism can be used, rather than in a secondary function (i.e., a second function called by the first C function invoked directly from Smalltalk).

The poKeep argument points to a single _oop object. This memory slot pointed to by poKeep will be updated by DLL and C Connect if the Smalltalk memory manager happens to move the location of the referenced object. If you need to store multiple object pointers you should construct a Smalltalk Array object using the oeAllocArray() function and use a pointer to this Array _oop as the poKeep argument.

The following code snippet is an example of storing several objects during a callback operation. The example stores the receiver, selector, and argument oops passed to the sendBack() function and extracts them before invoking the next callback.

```
#include "userprim.h"
_oop
sendBack(_oop oReceiver, _oop oSelector, _oop oArgs)
{
 oop oArray;
 int i;
 oArray = oeAllocArray(oReceiver, 3);
 oeBasicAtPut(oArray, 1, oTheReceiver);
 oeBasicAtPut(oArray, 2, oSelector);
 oeBasicAtPut(oArray, 3, oArgs);
 for (i=0; i < 10; i++) {
  _oop oTheReceiver = oeBasicAt(oArray, 1);
  _oop oTheSelector = oeBasicAt(oArray, 2);
  _oop oTheArgs = oeBasicAt(oArray, 3);
  oeSendMessageMany(
   oTheReceiver,
   oTheSelector,
   oTheArgs,
   &oArray,
   (_oop) 0);
 }
}
```

Each oeSendMessage...() function returns the value returned by the message-send callback. The last argument of oeSendMessageMany() can be used to detect a failed message-send. If the message-send fails (for example, “message not understood”), or if any argument is invalid, the function returns the value of the last argument. By following this convention, you may test for a failure by comparing the result of the call to oeSendMessageMany() with the value passed as the last argument (in this example code fragment, zero is passed as a known value). In the code fragment above, this test is not performed.

The _oop type is used for the function’s return value and each of its arguments. This data type is defined in the userprim.h file, located in the src subdirectory of the DLL and C Connect release, and must be included in your C code.

For a complete description of the function prototypes for external message-sends, see the chapter Object Engine Access Overview. As a summary, the external message-send functions are oeSendMessage0(), oeSendMessage1(), oeSendMessage2(), oeSendMessage3(), and oeSendMessageMany(). An example of a one-keyword message-sending function follows:

```
_oop
oeSendMessage1(
 _oop oReceiver,
 _oop oSelector,
 _oop oArg1,
 _oop *poKeep,
 _oop oFailure);
```

The oSelector argument must be an instance of Symbol. For example, a valid selector argument for oeSendMessage1 would be #add:. A parallel set of functions enables you to specify the selector name as a null-terminated sequence of characters. In effect, your C code can generate the selector name immediately instead of being restricted to the selector that is passed from Smalltalk. The differences between these functions which allow a C-string message selector and the foregoing set are as follows:

- The method names have a C in its prefix, as in oeCSendMessage1.
- The oSelector argument type is char * instead of _oop.

For example, a unary message-sending function can be typically defined as:

```
_oop
eCSendMessage0(
 _oop oReceiver,
  char *szSelector,
 _oop *poKeep,
 _oop oFailure);
```

For the sake of simplicity, this example merely demonstrates how to arrange an external message-send callback. The example ignores useful functionality on the C side, focusing instead on the required setup for executing a Smalltalk message-send in a C function.

We can also create a generic C function that accepts a receiver, a selector, and an array of arguments from Smalltalk. It then executes the message-send (by calling back into Smalltalk). The function is called `sendBack`, and it is defined as follows:

```
#include "userprim.h"
_oop
sendBack(_oop oReceiver, _oop oSelector, _oop oArgs)
{
 _oop oResult;
 oResult = oeSendMessageMany(
  oReceiver, oSelector, oArgs, (_oop) 0, (_oop) 0);
 if (oResult == (_oop) 0) oeFail(0);
 return oResult;
}
```

Notice that there is no need for the `poKeep` facility, so a placeholder is put in that position. To test this example, compile and link **`sendBack.c`** as a dynamic-link library, and define an interface class in Smalltalk that names **`sendBack.dll`** (or whatever convention is used on your platform) as the library file. On the instance side of the `SendBackInterface` class, create the following function-invoking method:

```
sendTo: receiver selector: selector withArgs: argArray
 <C: _oop sendBack(
  _oop receiver,
  _oop(Symbol) selector,
  _oop(Array) argArray)>
```

Naming the type of object that is expected (Symbol and Array, in this case) enforces type checking on the affected arguments.

To test this example, execute the following expression in a workspace. Select the expression and choose Print It or Inspect to see the returned value:

```
SendBackInterface new
 sendTo: 'Hello'
 selector: #,
 withArgs: #('world!')
```

Limitations of Callbacks

Thunking

Associated with each instance of a CCallback object is an invisible object called a thunk. A thunk is the mechanism that transitions the caller's C code to Smalltalk code. The CCallback and its associated thunk are freed when the CCallback is no longer referenced from the system. Because of this, your code must retain a reference to the CCallback object while the callback is in progress. In addition, any reference to the callback thunk that is maintained by your external code does not remain valid across snapshots. Normally, you would make a new CCallback object after restarting from a snapshot. If, for some reason, you wish to maintain a CCallback object over a long period of time, you should be aware that it will need to be reinitialized after each snapshot. You can manually reinitialize the external reference by sending the CCallback instance the message mapAddress.

The following diagram illustrates the thunk mechanism used to implement callbacks:

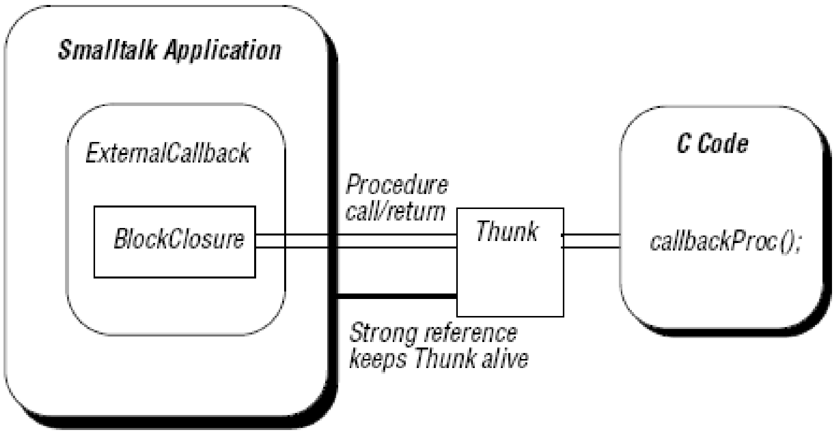


Figure 12: Implementing callbacks

An exception that unwinds the stack past a callback terminates it without the C code ever being aware of it. This also applies when a Smalltalk process is terminated while it is within a callback. This can produce a potential overflow condition on the C execution stack. This is a known limitation that will be corrected in a future release of DLL and C Connect.

Ordering of Callbacks

The current callback implements a last-in-first-out ordering restriction on all callbacks. That is, the last callback to make a transition from C into Smalltalk must be the first callback to return. It is possible to have application code make a callback where, while running within Smalltalk, a Smalltalk process switch occurs. The new process makes a C call that invokes a callback. If a process switch occurs back to the first process and it attempts to return, it blocks until the later callback completes. This last-in-first-out restriction introduces the possibility for deadlock.

This deadlock can be avoided by using the Threaded Interconnect interface instead of using simple callbacks. Note, however, that the Threaded Interconnect does not permit external message-sends from threaded calls. For Smalltalk applications which use multiple threads of control, the Threaded Interconnect is recommended instead of simple callbacks (for details, see the chapter Threaded Interconnect).

Valid Callback Locations

Another limitation on callbacks involves specifying the time when a callback may be invoked. Whether you are using Object Engine access functions `OE{C}SendMessage{0, 1, 2, 3, Many}` or `CCallback`, your callbacks can only be invoked from C code that was entered using the C calling machinery. Your C code should not invoke a callback from a user-primitive, from a signal handler, or from an interrupt handler.

Object Pointers

`CCallback` cannot be used to send object pointers (`_oop`) back to Smalltalk. Therefore, object pointers are invalid arguments for `CCallback`.

Chapter

5

Threaded Interconnect

Topics

This chapter explains how to use the DLL and C Connect Threaded API (THAPI) mechanism. This mechanism enables a VisualWorks application to make multiple concurrent, possibly blocking, calls to external code, each on its own independent thread of control. The Threaded Interconnect also allows VisualWorks to handle multiple concurrent callbacks from external code running on any thread under the control of the VisualWorks process.

The following discussion assumes that you have a general understanding of the strategies for concurrent processing in VisualWorks.

For a more detailed discussion of Smalltalk processes, semaphores, signals, and shared queues, you may consult the *Application Developer's Guide*.

Overview

Viewed as a complete package, DLL and C Connect provides two basic mechanisms for making calls to C functions, one synchronous, the other asynchronous. The standard DLL and C Connect interface only allows synchronous calls; that is, during a C function call, all Smalltalk processes being executed by the Object Engine are blocked for the duration of the call. By contrast, the Threaded Interconnect API allows for asynchronous calls.

For Smalltalk applications that make use of DLL and C Connect for heavy I/O operations, network streams, or concurrent processing with multiple threads of control, the Threaded Interconnect API is strongly recommended. For example, a middle-tier application server in a three-tier architecture should ideally be implemented using THAPI. By providing an asynchronous execution model, the DLL and C Connect Threaded Interconnect interface can yield a significant performance increase in the throughput of I/O processing.

Threads

To understand the asynchronous execution model employed by the DLL and C Connect threaded interconnect, it is useful to first distinguish between Smalltalk processes, OS processes, and threads.

On conventional operating-system platforms that provide their own multiprocessing model, the Smalltalk Object Engine runs as a single OS process with one thread of control. Since the memory overhead and performance demands for each OS process can be considerable, these are often referred to as heavyweight processes. Each OS process may also contain several independent threads of control (sometimes referred to as native threads).

As a “lightweight” alternative to OS processes, threads can yield improved performance and dramatically reduce resource usage. Smalltalk processes (STPs) are similarly lightweight in the sense that the Object Engine uses the process scheduler to manage each STP internally, multiplexing (sharing) the single thread of control between the various STPs, as appropriate.

The Object Engine's execution model is essentially equivalent to what is provided by an OS thread manager. However, multiprocessing in Smalltalk is invisible to other OS processes; from the point-of-view of the host OS, the Object Engine and all Smalltalk processes appear to combined as a single thread of control. This becomes an issue when the heavyweight Object Engine process performs I/O operations. The problem is that function calls routed through the standard DLL and C Connect interface may be blocked by the host OS (while waiting for I/O to complete), and this in turn can block the Object Engine from executing all other Smalltalk processes.

To eliminate this problem with blocking I/O, the Threaded Interconnect API provides a means to execute multiple native-code threads within the single OS process that runs the Object Engine. These threads are not mapped to each Smalltalk process run by the Object Engine, but are rather used only for I/O calls via the DLL and C Connect interface. From the DLL and C Connect programmer's point of view, the Object Engine still runs as a single thread, with the threaded interconnect calls running alongside it.

In this model, whenever a potentially blocking I/O call is to be made via DLL and C Connect, a separate thread is handed the information necessary to make the call, the call is executed, and is blocked by the OS until it completes and the OS reschedules the thread. Meanwhile, the Object Engine thread blocks the particular Smalltalk process that invoked the operation, scheduling other runnable STPs. Once the I/O thread is rescheduled by the OS, it passes back the result(s) of the call to the Object Engine thread. The Object Engine thread is then free to resume the STP that invoked the call. Once the I/O thread has passed its results back, it can either terminate, or remain in a quiescent state, awaiting subsequent requests for I/O operations.

The threaded execution model can be illustrated as follows:

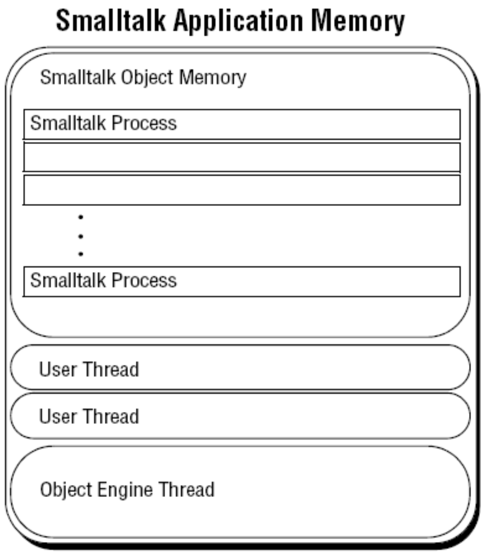


Figure 13: Threaded Execution Model

Threads and Synchonization

As a general rule, OS threads do not guarantee that changes to data in one thread become immediately visible to other threads. To ensure that such changes become visible, provisions like memory barrier instructions or OS locks are needed to synchronize data.

For some applications, this may be an issue, as when multiple Smalltalk processes make simultanous asynchronous calls using THAPI. I.e., if a first thread makes changes to some shared data, when would these be visible to a second thread?

In all VisualWorks implementations of THAPI (i.e., all platforms), if two calls are executed at the same time, it is the developer's responsibility to syncronize changes. However, if the first call finishes before the second one is started, all changes made by the first call are visible to the second.

VisualWorks implements passing of arguments and return values to THAPI calls by writing the data into global memory in the Smalltalk thread, using an OS lock or similar synchronization, and then using the data in the other thread. This means that all all modifications done by one call are immediately visible to any *subsequent* threaded call without any special provisions.

Managing Data Objects with Multiple I/O Threads

As discussed in the chapter Creating and Accessing C Data, special care must be taken to manage data objects that are passed to foreign code functions. The Object Engine routinely invokes the garbage collector to move objects within the heap. Although this can happen at arbitrary times, it is not a problem for temporary data objects being passed via synchronous calls. For the duration of a synchronous call, the garbage collector is blocked, along with the rest of the Object Engine, and hence no object relocation can occur.

However, moving objects within the heap is a problem if the call invokes code that tries to remember the address of an object for subsequent calls, or if the code invokes a callback. In either case, when the Object Engine runs, the garbage collector might move the object, invalidating the information retained by the foreign code.

The same situation arises when using the threaded interconnect. Since the Object Engine is not blocked during a threaded call, the garbage collector is free to move objects while the call is in progress. If the garbage collector moves an object being used as an argument to a threaded call, either invalid data will be passed to the foreign code, or the foreign code will likely write to invalid addresses and corrupt the Smalltalk heap.

It is the programmer’s responsibility to cope with foreign code that depends on being passed fixed pointers by using one of the storage allocation strategies provided by DLL and C Connect which enable Smalltalk programmers to manipulate data on the C heap and reference this data using instances of `CPointer`.

For these reasons, all reference parameters in threaded calls must refer to data that does not move for the duration of the call. There are three ways to achieve this:

- Use `malloc` to allocate storage on the C heap, and then explicitly free the C data object later.

- Copy the object's data to some fixed space at the start of the call, and copy it back once the call completes.
- Relocate the object's data to a fixed space, where it resides for at least the duration of the call.

The second alternative does not preserve referential integrity; that is, if an object is passed to more than one concurrent threaded call, or if Smalltalk code accesses the object while the call is in progress, the participants see different copies of the data. This is a serious problem in applications that need to share data (e.g. applications that share pools of I/O buffers between I/O drivers and applications, as supported by some file systems such as Windows NT). Further, the overhead of copying data to and from some fixed space might be unacceptably high.

If the object might be passed concurrently to more than one threaded call, the preferred strategy is to use the special storage area in the Smalltalk heap called `FixedSpace`. The bodies (contents) of objects in `FixedSpace` do not move, preserving referential integrity and allowing them to be used freely as the arguments in threaded (and normal) calls. Promotion of objects to `FixedSpace` is automatic, and occurs when a mobile argument is passed as an argument to a threaded call. For a more detailed discussion of `FixedSpace`, see: Allocating Objects in FixedSpace.

Threaded Interconnect Example

From the programmer's perspective, the Threaded Interconnect is a relatively small extension of the standard DLL and C Connect interface.

A good way to explain the threaded calling mechanism is through a simple example that illustrates all features except callbacks. In this example, a multithreaded "server" is constructed with a number of Smalltalk processes waiting for "requests" on an I/O connection. For simplicity, the requests within a single Smalltalk image are generated and served.

In this example, a "request" is simply a character string, and a response to a request is to display the string in the System Transcript. Requests are written to a single pipe. (A pipe is an I/O channel represented by a pair of file descriptors. A write of data via the write file descriptor makes that data available for reads via the read file descriptor.) Each “server” process is a loop that blocks, waiting for data to appear on the pipe before writing this data to the transcript. Another process makes “requests” by writing data to the pipe. To give the Smalltalk system something computationally intensive to do while all this is going on, the data written to the pipe is the result of running a simple benchmark.

Our example class is NonBlockingPipeInterface, since it waits for data on a pipe without blocking the Object Engine. It is a subclass of ExternalInterface, since it has a number of external methods. Two instance variables, infd and outfd, are file descriptors for the pipe, while the boolean flag running determines when each “server” process terminates. The source for this example can be found in the parcel **THAPIExample.pst** that is included in the DLL and C Connect subdirectory that accompanies the VisualWorks release.

Each “server” loops in the reader: method, reading data and then writing the data to the Transcript.

```
reader: id
 "Read from the pipe as long as running is true.
 Print whatever is read from the pipe to the Transcript and
 tag it with id."
 | buffer count |
 "Use a buffer on the C heap for the read call."
 buffer := CIntegerType char gcMalloc: 1025.
 [running] whileTrue:       "Continue until running is false"
 "Make a blocking read from pipe on its own thread."
 [count := self read: infd with: buffer with: 1024.
 "Use a mutex to serialize writing to Transcript"
 TranscriptProtect critical:
  [Transcript cr; print: id; tab. "Print this reader's id tag."
  "Check read result and complain if its in error."
  (count > 1024 or: [count < 0])
   ifTrue: [Transcript nextPutAll: 'READ RETURNED ';
          print: count]
   ifFalse:
    ["Null-terminate then copy data as a String."
    buffer at: count put: 0.
    Transcript nextPutAll: buffer copyCStringFromHeap].
  Transcript endEntry]]
```

The read buffer is allocated on the C heap. The buffer is 1025 bytes long, large enough for 1024 characters and a null-terminating byte. Smalltalk objects can be moved by the garbage collector, which might run while a threaded call is in progress. Consequently, Smalltalk object pointers cannot be passed as arguments to `_threaded` calls. Later, this example is refined to include the use of objects allocated in `FixedSpace`.

The Transcript is not thread safe, so access to multiple reader processes attempting to write to the Transcript at the same time must be serialized. A class variable, `TranscriptProtect`, is used to achieve this. This variable is defined in a class initialization method.

```
initialize
 "Initialize the mutex for serializing writes to the Transcript
 and a constant to open the pipe in binary mode."
 TranscriptProtect := Semaphore forMutualExclusion.
 "initialize a class variable for use with NT pipe mode argument."
 O_BINARY := 16r8000        "from msdev\include\fcntl.h"

 "self initialize"
```

Specifying Threaded External Methods

As for the threaded call itself, the message send that invokes the call is indistinguishable from a normal DLL and C Connect call. The threaded-ness is a property described in the external method specifying the call. Threaded calls are specified by using the `_threaded` pseudo-qualifier in the C pragma of your C function prototype definition. Note that the `_threaded` keyword must follow the function's return-type. The type of the buffer argument is `_oopref *`, which passes a pointer to the buffer's contents without interpreting its contents. (Using a type such as `char *` causes the contents of a `ByteString` or `TwoByteString` buffer to be checked to ensure that the character set in the string agrees with the platform's character set, which is unnecessary in this example.)

Thus, to specify a threaded call in the example interface class, the declarative method would be as follows:

```
read: fd with: buffer with: size
 "Invoke the read system call on its own thread and hence avoid
```

```
blocking the Object Engine."
<C: long _threaded read(int fd, _oopref *buffer, unsigned long size)>
^self externalAccessFailed
```

An invocation of the method causes the calling Smalltalk process to block until the read call returns. Meanwhile, other runnable Smalltalk processes can execute. To perform the call, the Object Engine provides a thread that is available to make the call, passes all the information necessary to make the call (the function and arguments) to the thread, and then blocks the calling Smalltalk process until the thread returns a result. Once the result is returned, the Object Engine passes the result back to the process and allows it to continue.

On most operating systems, the thread making the call is given the next higher priority to the Object Engine thread to ensure that it makes progress, even if the Smalltalk system has other runnable processes for the Object Engine to execute. On Linux and HPUX, the thread has the same priority as the Object Engine.

Since pipes have limited capacity, it is possible to block when writing to a pipe. A write to a pipe might block until a sufficient number of reads have been done to make space available for the write. Thus, to ensure that the example's computation is not interrupted by potentially blocking writes, a separate process is used to perform the writes via threaded calls. The write process reads results from the generator through an instance variable, `results`, which holds an instance of `SharedQueue`. Class `SharedQueue` provides a thread-safe way of communicating between processes, somewhat analogous to a pipe. An object added to the queue via `nextPut:` is available via `next`. If the queue is empty, the calling process blocks in the `next` method until an object is added to the queue via `nextPut:`.

The `writer` method is rather similar to the reader:

```
writer
 "Loop writing strings from the results queue to the pipe."
 | result buffer writeCount |
  "Use a buffer on the C heap for the read call."
 buffer := CIntegerType char gcMalloc: 1024.
 [true] whileTrue:
  ["Get the next result from the results shared queue.
  This process waits until one is available. Convert the result
  to a ByteArray since the buffer is of type char (an integer)."
  result := results next asByteArray.
  "Copy the string into the buffer."
  buffer copyAt: 0 from: result size: results size startingAt: 1.
  "Write the buffer's data to the pipe."
  writeCount := self write: outfd with: buffer with: results size.
  "Check the write operation succeeded."
  writeCount ~= results size
   ifTrue: [TranscriptProtect critical:
      [Transcript
      cr;
      nextPutAll: 'WRITE RETURNED ';
      print: writeCount;
      nextPutAll: ' EXPECTED ';
      print: results size;
      nextPut: $!; endEntry]]]
```

The writer does not test the boolean flag `running`, because it is explicitly terminated.

The method for performing the write is also a threaded call:

```
NonBlockingPipeInterface methods for procedures

write: fd with: buffer with: size
 "Invoke the write system call on its own thread and hence
 avoid blocking the Object Engine."
 <C: long _threaded write(int fd, _oopref *buffer, unsigned long
  size)>
 ^self externalAccessFailed
```

To implement the rest of the interface, we first need some interface functions to open and close the pipe, and we need a benchmark to run.

```
close: fd
 "Close the file descriptor fd"
 <C: int close(int fd)>
 ^self externalAccessFailed

pipe: arg
 "UNIX pipe creation function."
 <C: int pipe(int [])>
 ^self externalAccessFailed

pipe: arg ofSize: size mode: textMode
 "NT pipe creation function."
 <C: int pipe(int arg[], unsigned int size, int textMode)>
 ^self externalAccessFailed

Integer methods for mathematical functions

nfib
 "The nfib benchmark calculates a rough measure of activations
 per second. This is a version of fibonacci in which 1 is added for
 each activation. The result is therefore equal to the number of
 activations required to calculate that result. To get the 'nfib'
 figure of nfib activations per second choose a value which takes
 nfib about 30 seconds to calculate. Then divide the result by the
 time taken, yielding activations per second."
 self < 2
   ifTrue: [^1]
   ifFalse: [^(self - 1) nfib + (self - 2) nfib + 1]
```

A separate method is used to open the pipe, because it must be opened differently under UNIX and Windows. Also, a separate method is used to terminate the example, because it might be run in the background and need to be terminated from some other process.

```
openPipes
 "Create the pipe. Note that pipe in the MSVC run-time library
 is different from the standard UNIX pipe."
 | fds |
 fds := CIntegerType int gcMalloc: 2.
 (OSHandle currentOS == #win32
   ifTrue: [self pipe: fds ofSize: 1024 mode: O_BINARY]
   ifFalse: [self pipe: fds]) < 0
  ifTrue: [self error: 'pipe open failed.'].
 infd := fds at: 0.
 outfd := fds at: 1
```

The `terminate` method is careful to do nothing if already terminated. On process termination, any unwind blocks in the process are run. Hence, if `terminate` is sent from another process, it gets sent again from the unwind block in the `readers:` method when terminate kills the generator process. The instance variable `generator` is used to refer to the process running the benchmark, and the instance variable `readers` is used to refer to the collection of readers.

```
terminate
 "Terminate all the relevant processes and close the pipe."
 running ifFalse: [^self]. "Do nothing if already terminated."
 running := false.
 generator == Processor activeProcess
       ifFalse: [generator terminate].
 "Write sufficient data to the pipe so that all readers get data,
 and hence by checking running, stop."
 readers size * 2 timesRepeat: [results nextPut: 'so long!'].
 "Delay until the results have been written by the writer and then
 kill the writer. Yield doesn't work if the process doing terminate
 has a higher priority than the writer so use a delay."
 [results isEmpty] whileFalse: [(Delay forMilliseconds: 20) wait].
 writer terminate.
 "Close the pipe"
 self close: infd; close: outfd.
```

The `readers:` method serves as the main loop of the example. It opens the pipe, creates a shared queue to communicate results to the writer, spawns the readers and writer, and then loops, generating data. On unwind, it calls `terminate` to shut down.

```
readers: n
 "Run the example with n reader processes."

 results := SharedQueue new.
 self openPipes.
 "remember the generator process for terminate."
 generator := Processor activeProcess.
 "running is polled by other processes so they'll terminate
 when we're done."
 running := true.

 "Fork a writer process to write data to the pipe. Its priority is
 higher than the generator to ensure writes happen promptly."
 writer := [self writer] forkAt: generator priority + 1.

 "Fork n readers at a higher priority so that results get read
 and displayed."
```

```
readers := (1 to: n)
 collect: [:i | [Processor yield. self reader: i]
  forkAt: generator priority + 1].

"Generate some data using the example benchmark."
[ | i r t s nfibs |
 s := String new writeStream.
 i := 0.
 [t := Time millisecondsToRun: [r := i nfib].
 s reset.
 nfibs := Number errorSignal
  handle: [:ex | '??']
  do: [(r * 1000.0 / t) rounded].
 s nextPutAll: 'nfib '; print: i; nextPutAll: ' = '; print: r;
  tab; tab;
  nextPutAll: 'nfibs '; print: nfibs;
  nextPutAll: ' ('; print: t / 1000.0; nextPutAll: ' seconds)'.

 "Put datum in results shared queue for the writer to
 consume."
 results nextPut: s contents.

 "Increase the value from which we compute nfib, limiting it
 to one that takes 30 seconds or less to run."
 i := t > 30000 ifTrue: [0] ifFalse: [i + 1]] repeat] valueNowOrOnUnwindDo: [self
terminate]
```

The underlying C functions for accessing the pipe are in a C library. On Windows the C library is one of the `MSVCRTnn.DLL` DLLs, while on Solaris the C library is `/usr/lib/libc.so`. You can use a single interface class for all these cases, provided the interface copes with the `libraryNotFoundSignal`, which is raised when an attempt is made to open a nonexistent library. For example, the `libraryNotFoundSignal` signal is raised if the interface tries to open `/usr/shlib/libc.so` on a Windows machine.

`ExternalInterface` supports a standard idiom for doing just this. The class declaration should include the full set of library files and directories for all systems, and on the class side of the interface you implement the `libraryFilesSearchSignals` method to return the `Signal` or `SignalCollection` of the signals to be ignored during library loading. Thus, you need the following method to avoid raising a signal when using the interface’s procedures:

```
libraryFilesSearchSignals
 "Answer a SignalCollection used to handle exceptions raised
 when scanning for library files. The signals answered by this
 method results in those signals being ignored by the library
 search machinery. Clients should not answer signals they wish
 to receive."

 ^ExternalLibraryHolder libraryNotFoundSignal
```

On Windows you also need to know where to look for the library **`MSVCRTnn.DLL`**. Here you can make use of environment variables in the list of library files and directories. (You can also use patterns to match against `OSHandle currentPlatformID`. Browse `ExternalLibrary >> findFile:inDirectories:` for a full description.) For a more detailed discussion, see Dynamic-Link Libraries, “Libraries and Environment Variables.”. In the following, **`$(windir)`** expands to the value of the **`windir`** environment variable; for example, **`C:\Windows`**.

The following class declaration loads the appropriate C library on Windows NT 3.51 & 4.0, as well as Solaris and Digital UNIX.

```
Smalltalk.Examples defineClass: #NonBlockingPipeInterface
 superclass: #{External.ExternalInterface}
 indexedType: #none
 private: false
 instanceVariableNames: 'infd outfd generator writer readers
  results running '
 classInstanceVariableNames: ''
 imports: '
   private Examples.NonBlockingPipeInterfaceDictionary.*
    '
 category: 'ExternalInterface-THAPI Example'
 attributes: #(
   #(#includeFiles #())
   #(#includeDirectories #())
   #(#libraryFiles #('libc.so' 'libc.sl' 'msvcrt40.dll'))
   #(#libraryDirectories #('[unix]/usr/shlib' '[unix]/usr/lib'
     '[win]$(windir)\system' '[win]$(windir)\system32'))
   #(#beVirtual false)
   #(#optimizationLevel #full))
```

To run the example, evaluate the following expression:

```
NonBlockingPipeInterface new readers: 10
```

The following Transcript output was taken from executing this example on a 60MHz Pentium-class machine running MS-Windows:

```
1 nfib 0 = 1     nfibs '??' (0.0 seconds)
2 nfib 1 = 1     nfibs '??' (0.0 seconds)
3 nfib 2 = 3     nfibs '??' (0.0 seconds)
4 nfib 3 = 5     nfibs '??' (0.0 seconds)
5 nfib 4 = 9     nfibs '??' (0.0 seconds)
6 nfib 5 = 15     nfibs '??' (0.0 seconds)
7 nfib 6 = 25     nfibs '??' (0.0 seconds)
8 nfib 7 = 41     nfibs '??' (0.0 seconds)
9 nfib 8 = 67     nfibs 67000 (0.001 seconds)
10 nfib 9 = 109     nfibs '??' (0.0 seconds)
1 nfib 10 = 177     nfibs '??' (0.0 seconds)
2 nfib 11 = 287     nfibs '??' (0.0 seconds)
3 nfib 12 = 465     nfibs 465000 (0.001 seconds)
4 nfib 13 = 753     nfibs '??' (0.0 seconds)
5 nfib 14 = 1219     nfibs 1219000 (0.001 seconds)
6 nfib 15 = 1973     nfibs '??' (0.0 seconds)
7 nfib 16 = 3193     nfibs '??' (0.0 seconds)
8 nfib 17 = 5167     nfibs 5167000 (0.001 seconds)
9 nfib 18 = 8361     nfibs 4180500 (0.002 seconds)
10 nfib 19 = 13529     nfibs 3382250 (0.004 seconds)
[...]
```

This process runs until it is interrupted using Control-C, and when the notifier is terminated, the processes stop running, indicating their completion with the message “`so long!`” in the `Transcript` window.

Callbacks

The Threaded Interconnect supports callbacks from arbitrary threads. Callbacks are not restricted merely to threads used for threaded callouts. However, the exact handling of callbacks does depend on which thread makes the callback. Threaded callbacks do not require any new syntax. The threaded-ness of a callback is determined by which kind of thread calls back.

First, if a callback is made by the Object Engine thread, as happens when a normal (not threaded) callout calls back, the callback is handled synchronously. The last-in, first-out restriction on such callbacks is still present. For a more detailed discussion of synchronous callbacks, see Calling Smalltalk From C.

If a callback happens from a thread used to make a threaded callout, then the callback runs in the same process that made the threaded callout. Consequently, it runs at the same priority as the process that made the callout. If the process makes a nested callout (i.e., if it performs a threaded callout within a threaded callback), then the callout happens on the same thread.

A callback from any other thread is termed a foreign callback, since the thread is not in the Object Engine's thread pool. Foreign callbacks run on their own special process. A new Smalltalk process is created to run each bottom-level foreign callback. The priority of this new process is controlled by the `ForeignCallbackPriority` static in `CCallback` and by default is `Processor lowIOPriority - 5`. No protocol is available to change this priority.

You can change the priority explicitly, using the following code fragment:

```
CCallback classPool
 at: #ForeignCallbackPriority
 put: Processor userInterruptPriority - 1
```

The best technique is to set the priority explicitly within the Smalltalk code for the callback. You can find an example in the code for class `CCallback`.

Once a foreign callback occurs, the process created to service the callback remains associated with that thread until the callback returns (or the process terminates). Subsequent threaded callouts from this process happen in the foreign thread. If one of these callouts were to call back, then the callback would run on the same process. Hence, a foreign callback process is created only for each bottom-level foreign callback.

Since each threaded callback (foreign or otherwise) is running on its own thread's stack, there is no problem with the ordering of returns from threaded callbacks. The LIFO restriction on normal callbacks results from the fact that each callback shares the stack of the Smalltalk thread, so each must return in LIFO order to ensure the stack can be cut-back safely.

Additional Control over Threads

Managing Threads

Because the cost of thread creation can be high, the Threaded Interconnect actually manages a pool of threads, rather than creating a new thread for each threaded call. (For example, under Windows NT 4.0, the overhead of creating a new thread for a call adds close to 1100 percent of the normal time duration for the call.)

Normally, THAPI manages the thread pool automatically, but there are at least two circumstances under which the programmer may need to manage the allocation of threads explicitly. THAPI provides a special protocol for controlling thread allocation. These circumstances are as follows.

First, to avoid certain types of deadlock in multithreaded applications, it may be necessary to attach Smalltalk processes to specific threads. And second, because some applications require that a native code API be used by a specific thread, the interconnect also allows programmers to ensure that a specific thread is used to make a call. The Threaded Interconnect accomplishes this by maintaining a pool of active threads, and, as much as possible, by using the same I/O thread to perform calls on behalf of a specific Smalltalk process.

In this section, mechanisms are described for controlling the size of the thread pool, and for reserving an I/O thread for the sole use of a specific Smalltalk process.

Thread Limit and Low Tide

Under certain circumstances, it is possible to deadlock a multithreaded application due to a condition known as thread starvation. To avoid this condition, the programmer must exercise some control over the allocation of threads in the pool. After considering the mechanisms for controlling the number of threads that can be created, we shall illustrate their use by extending the `NonBlockingPipeInterface` example.

THAPI provides a hard limit on the maximum number of threads that can be created and a low-tide limit on the number of quiescent threads. When the Object Engine starts up, it initializes both the upper and lower thread limits to 32.

A thread is created when a threaded call is made by a process with no associated thread and no unassociated threads exist in the pool. For the duration of the call, the thread is then associated with the calling process and is used in any nested threaded calls. For example, if a callback occurs during the call, the callback runs in the same process that made the callout. Any threaded callouts made from this process while the outermost threaded call is still in progress are made by the same thread.

Once the outermost threaded call returns, the thread is disassociated from the calling process and is returned to the pool. The thread can then be used to perform a call on behalf of any process. Thus, a burst of concurrent threaded calls can result in the creation of a number of threads which, when the calls return, end up unassociated in the pool. The live thread low tide is used to control the size of the pool. If the total number of threads maintained by Object Engine exceeds the low tide, then unassociated threads in the pool are terminated until the low tide is met.

For example, if we set the limit to 64 and the low tide to 32 and evaluated `NonBlockingPipeInterface new readers: 60`, then the Object Engine creates 61 threads (60 concurrent read calls from readers and one concurrent write call from the writer). As each call returns, the thread that made the call is terminated, because the total number of threads is higher than the low tide. However, once the example terminates and all calls return, the Object Engine keeps 32 threads alive, ready for future use.

The following table describes the low-level methods which provide access to the Object Engine settings for the thread limit and low tide value.

Table 18: Protocol for managing Object Engine thread limits

Method	Description
ProcessorScheduler >> primGetThreadLevels	Return an array of seven elements: @1 is the limit on the number of active threads created by the OE. @2 is the low tide on the number of active threads maintained by the OE. The OE does not reduce the number of inactive threads below this level. @3 is the current number of active threads. @4 is the current number of inactive (but live) threads. @5 is the number of foreign threads currently calling in. @6 is the number of threads created by the OE since start-up. @7 is the number of threads killed by the OE since start-up. Counts 1 through 4 are inclusive of the Object Engine's single Smalltalk thread. Fail if the array cannot be created.
ProcessorScheduler >> primSetThreadLimit: limit lowTide: lowTide	Set the thread limit and low tide. A less-than-zero value is ignored, allowing each level to be set independently. Note that these values must be at least 1 because there is one thread devoted to Smalltalk.

Attaching Processes to Threads

If you try to run the NonBlockingPipeInterface example with more readers than the live thread limit, the example deadlocks. As readers are forked, they make threaded calls, resulting in threads being created. Once the number of readers exceeds the thread limit, subsequently attempted read calls will raise an externalAccessFailedSignal that indicates an "out of threads" error. When the writer attempts its write call, it fails in the same way for the same reason. Thus, to guarantee progress in the presence of potential thread starvation, a thread must be reserved for exclusive use by the writer. This is called attaching a process to a thread. Another circumstance in which you need to attach a process to a given thread is where a specific thread must be used to make certain calls (one example is in Windows, where a specific thread must be used to make debugging calls to inspect the state of a process being debugged).

The following table describes the methods that provide control over thread attachment.

Table 19: Controlling the attachment of threads to Smalltalk processes

Method	Description
Process>>attachToThread	Reserve a native thread for the receiver to make _threaded calls.
Process>>detachFromThread	Release the native thread from its attachment to the receiver.
Process>>isAttachedToThread	Answer whether the receiver is attached to a native thread.

To illustrate these protocols for thread control, the NonBlockingPipeInterface example will be extended to cope with thread starvation. To do this, we must change the readers: method to reserve a thread for the writer, and add a method called startReader: that will handle out-of-threads exceptions in reader processes, as follows:

```
NonBlockingPipeInterface methods for public access

readers: n
 "Run the example with n reader processes."
 results := SharedQueue new.
 self openPipes.
 "remember the generator process for terminate."
 generator := Processor activeProcess.
 running := true.
 "Fork a writer process to write data to the pipe. Its priority is
 higher than the generator to ensure writes happen promptly.
 It needs to reserve its own thread because there might be
 more readers than available threads, and if the writer can't get
 a thread because they have been used up by the readers the
 example deadlocks, since the readers get no data unless some
 is written."
 writer := [self writer] forkAt: generator priority + 1.
 writer attachToThread.
 "Fork n readers at a higher priority so that results get read and
 displayed."
 readers := (1 to: n) collect:
        [:i | [Processor yield. self startReader: i]
            forkAt: generator priority + 1].
 "Generate some data using the example benchmark."
 [ | i r t s nfibs |
  s := String new writeStream.
  i := 0.
 [t := Time millisecondsToRun: [r := i nfib].
 s reset.
 nfibs := Number errorSignal
  handle: [:ex | '??']
  do: [(r * 1000.0 / t) rounded].
 s nextPutAll: 'nfib '; print: i; nextPutAll: ' = '; print: r;
  tab; tab;
  nextPutAll: 'nfibs '; print: nfibs;
  nextPutAll: ' ('; print: t / 1000.0; nextPutAll: ' seconds)'.
 "Put datum in results shared queue for the writer to consume."
results nextPut: s contents.  "Increase the value from which we compute nfib, limiting
it
 to one that takes 30 seconds or less to run."
 i := t > 30000 ifTrue: [0] ifFalse: [i + 1]] repeat]
valueNowOrOnUnwindDo: [self terminate]
```

And here is a method that handles out-of-threads exceptions:

```
startReader: id
 "Place an exception handler around the reader: method to catch
 out-of-thread exceptions."
 self externalAccessFailedSignal
  handle:
   [:ex |
   ex name == #'out of threads'
    ifTrue:
     [TranscriptProtect critical:
      [Transcript
       cr; print: id;
       nextPutAll: ' CANNOT PROCEED. OUT OF
        THREADS';
       endEntry].
    Processor activeProcess terminate]
   ifFalse: [ex reject]]
  do: [self reader: id]
```

To test the new example, evaluate the following:

```
Processor primSetThreadLimit: 6 lowTide: 4.
NonBlockingPipeInterface new readers: 10
```

The limit is set to six threads. Inclusive of the Object Engine thread, this leaves five threads for making threaded calls. One is reserved for the writer thread. Thus, the last six readers fail due to the thread limit being exceeded, and they terminate, leaving four threads to read the pipe.

This thread limit is illustrated with the following excerpt from the System Transcript as the example is evaluated:

```
5 CANNOT PROCEED. OUT OF THREADS
6 CANNOT PROCEED. OUT OF THREADS
7 CANNOT PROCEED. OUT OF THREADS
8 CANNOT PROCEED. OUT OF THREADS
9 CANNOT PROCEED. OUT OF THREADS
10 CANNOT PROCEED. OUT OF THREADS
1 nfib 0 = 1     nfibs 1000 (0.001 seconds)
2 nfib 1 = 1     nfibs '??' (0.0 seconds)
3 nfib 2 = 3     nfibs '??' (0.0 seconds)
4 nfib 3 = 5     nfibs 5000 (0.001 seconds)
1 nfib 4 = 9     nfibs '??' (0.0 seconds)
2 nfib 5 = 15    nfibs '??' (0.0 seconds)
3 nfib 6 = 25    nfibs '??' (0.0 seconds)
4 nfib 7 = 41    nfibs '??' (0.0 seconds)
1 nfib 8 = 67    nfibs '??' (0.0 seconds)
[...]
```

Once again, the process runs until it is interrupted using Control-C, and when the notifier is terminated, the processes stop running, indicating their completion with the message “so long!” in the Transcript window.

Threaded Calls and FixedSpace

As discussed in the chapter Creating and Accessing C Data, special care must be taken when passing data objects as parameters to threaded calls. Since the garbage collector moves both object headers and object bodies when it collects, both oops (pointers to headers) and body pointers used with external code may become invalid. This imposes the restriction that you cannot pass object pointer arguments through threaded calls. Further, you cannot use the external message sending facilities (for details, see External Messages) from within threaded calls, because these also depend on object pointers.

The first strategy for allocating C data objects that can be shared with Smalltalk methods involves using the external heap (accessed via CDatum and its subclasses). However, using this strategy is often clumsy and can be inefficient, because data might have to be copied from the C heap into a Smalltalk object before it can be used.

To circumvent these problems, objects passed to threaded calls are typically allocated in FixedSpace, a special zone in the Smalltalk object memory (for a more extensive discussion of FixedSpace, see Allocating Objects in FixedSpace). The Object Engine ensures automatically that the body of any byte-like object that is passed as a pointer argument to a threaded call gets promoted to FixedSpace. This ensures that the garbage collector does not move the object's body during the _threaded call, although the garbage collector might move the object's header (and hence need to change its object pointer).

Using FixedSpace, we can now simplify the NonBlockingPipeInterface example by reimplementing the reader: method to use FixedSpace in a new subclass, FSNBPI (for FixedSpaceNonBlockingPipeInterface).

```
FSNBPI methods for server

reader: id
 | buffer count |
 "Use a normal Smalltalk string, but allocate it in FixedSpace."
 buffer := String defaultPlatformClass newInFixedSpace: 1024.
 [running] whileTrue:
  [count := self read: infd with: buffer with: buffer size.
  TranscriptProtect critical:
   [Transcript cr; print: id; tab.
   (count > buffer size or: [count < 0])
    ifTrue: [Transcript
        nextPutAll: 'READ RETURNED ';
        print: count]
    ifFalse: [1 to: count
        do: [:i | Transcript nextPut: (buffer at: i)]].
   Transcript endEntry]]
```

The writer method must also be reimplemented, as follows:

```
writer
 "Loop writing strings from the results queue to the pipe."
 | result writeCount |
[true] whileTrue:
 [result := results next.
 writeCount := self write: outfd with: result with: result size.
 writeCount ~= result size ifTrue:
  [TranscriptProtect critical:
   [Transcript
    cr;
    nextPutAll: 'WRITE RETURNED '; print: writeCount;
    nextPutAll: ' EXPECTED '; print: result size;
    nextPut: $!; endEntry]]]
```

Thus, the buffer in the above version of `reader:` is instantiated in `FixedSpace`, while in the above version of `writer` each string gets promoted to `FixedSpace` on each call of `write:with:with:`.

Limitations

Thread-Safety of Foreign Code

Any code called through the Threaded Interconnect that might be in use by more than one thread at the same time must be thread safe. For example, if you are trying to use THAPI to provide asynchronous database connectivity using the EXDI (External Database Interface), you should first research whether the database vendor’s client libraries are thread safe. Errors caused by using non-thread-safe code in a multithreaded context can be both difficult to find and potentially disastrous. In the best case, these kinds of problems might only crash the system or provoke an exception. In the worst case, they could result in data corruption. Further, the effects of these kinds of errors might show up long after the errors themselves occur.

For example, if you wanted to use Oracle with their OCI client library via THAPI, you must use version 7.3.2.2 or later. Prior to the Oracle 7.3.2.2 release, Oracle client libraries were not thread safe. Oracle 7.3.2.2 and onward client libraries are supposed to be thread safe, but this has not been fully tested in a VisualWorks or VisualWave environment. (Be careful to verify thread safety, even with vendor sources. Vendors have reported, for example, that while **`dbLib`** is not thread safe, **`CTLib`** is. But engineers have reported that even **`CTLib`** is not thread safe.)

Use of Object Pointers and Message Sends

As explained earlier in the section Threaded Calls and FixedSpace, you cannot pass object pointer arguments to threaded calls nor send external messages from within threads. This is because the garbage collector moves objects and hence updates oops at arbitrary times relative to other threads.

Thread Priority

All threads created by the Threaded Interconnect run at the next highest priority to the Object Engine thread. This is to ensure that they make progress relative to the Object Engine. In cases where threads are to be used to perform some blocking action, as in the example, this behavior is appropriate. However, if threads are to be used to perform some lengthy computation (i.e., they do not yield control to other threads), they prevent the Object Engine from running unless the underlying machine is a multiprocessor. It is possible to augment THAPI with control over thread priorities, but the impact on performance (that of potentially changing the priority of a thread on each call) outweighs the utility of priority control. There is nothing to prevent you from changing the priority of a thread within your own code, reached through a threaded call. Under UNIX, the Threaded Interconnect relies on the implementation of POSIX threads. On Windows, THAPI uses standard Windows threads.

For example, to change thread priority on Solaris, use code similar to the following:

```
priority = pthread_getprio(pthread_self());
if (pthread_setprio(pthread_self(),priority + delta) <= 0)
 errinfo = errno;
```

On Windows, use code such as the following:

```
priority = GetThreadPriority(GetCurrentThread());
if (!SetThreadPriority(GetCurrentThread(),priority + delta))
 errinfo = GetLastError();
```

Be aware that you should reset the thread's priority before returning, because the Threaded Interconnect does not alter a thread's priority, once created. Remember that if the Object Engine is busy, lower priority threads can be blocked indefinitely.

Maximum Number of Threads

The maximum number of threads active at any one time depends on the underlying platform. Threads created by THAPI have a default stack size, an operating system semaphore, and a small amount of memory for argument marshalling. The example presented in this chapter has been run with over a thousand readers/threads on Windows NT 4.0. An empirical test suggests that on a 40-Mbyte machine, the limit is around 1980 threads on Windows NT 4.0.

Performance Considerations

Threaded calls are considerably slower than normal, synchronous DLL and C Connect calls. For example, here are some times for making a zero-argument call that returns immediately, using a normal DLL and C Connect call, a threaded call, and a threaded call that involves creating a thread. These times were measured on a 60 MHz Pentium COMPAQ 5/60M running Windows NT 4.0.

- Normal DLL and C Connect Call: 5 microseconds
- Threaded DLL and C Connect Call: 310 microseconds
- Threaded DLL and C Connect Call with thread creation: 3400 microseconds

These measurements suggest that a threaded call is approximately 60 times slower than a regular synchronous call, while a threaded call that must create a new thread is almost 680 times slower. On Solaris and Digital UNIX the ratios differ considerably, because the underlying scheduler gives threads a more sluggish response. Note that although these ratios may seem dramatic, the overall increase in I/O throughput that can be gained by using threaded calls from a Smalltalk application will offset the performance hit on each threaded call.

Another factor that can impact system performance involves the scheduling of the Object Engine thread. The Object Engine sleeps if it has no runnable process, and this is an activity that can be time consuming. If a trivial threaded call is made, during which the

Object Engine goes to sleep, the call takes considerably longer, since the engine must wake up before the call can return. On the above machine, if the same trivial call is made when the engine sleeps (i.e., a call that does not require that a new thread be created), the call takes approximately 2400 microseconds (roughly eight times slower than a regular threaded call).

Sleeping can be avoided by providing a background process, or by keeping the system generally active. Thread creation can be avoided by setting appropriate thread limits and low tides. Profiling shows that the fundamental cost of a threaded call is due to the operating system scheduling facilities required by the internal design of THAPI. For each call, the following sequence of actions occurs to perform a threaded callout:

1. A signal from an operating system semaphore causes a thread switch (from the Object Engine Thread to the User Thread) to allow the thread to progress to make the call. The overhead involved in this action is probably minor.
2. On Windows a wakeup message is sent, and on UNIX a software signal is sent from the thread when it is ready to return results. The overhead involved in this action is probably rather significant.
3. The thread waits on an operating system semaphore, because it is now done, and this results in a thread switch back to the Object Engine thread.
4. Finally, the Object Engine performs a wakeup in response to the message or signal. The overhead involved in this action is probably rather significant.

Known Problems

Process Termination

Terminating a process that is waiting for the results of a threaded call kills the associated thread, unless the thread is foreign. This can create problems, depending on the context. For example, the following version of `terminate` (see below) works fine under Windows, but on Solaris it causes the Object Engine to freeze (because of a bug in Solaris). On Solaris, if any thread is in a `read()` call and some other thread closes the read side of the pipe, the process freezes. Each reader thread might not be terminated until reaching a termination point, so even though the Object Engine has attempted to terminate the reader threads, they might not terminate immediately, and if a thread remains in the read call when the read side of the pipe is closed, the entire process freezes.

```
NonBlockingPipeInterface methods for initialize-release

terminate
 "Terminate all the relevant processes and close the pipe."
 running ifFalse: [^self].
 running := false.
 generator == Processor activeProcess
      ifFalse: [generator terminate].
 writer terminate.
 readers do: [:ea | ea terminate].
 self close: infd; close: outfd
```

In general, it is wise to avoid terminating processes that have threaded calls in progress, unless absolutely necessary.

Chapter

6

Exception Handling

Topics

The reliability of your application depends largely upon how well it handles exceptions. DLL and C Connect performs extensive error checking to ensure that Smalltalk applications can interact smoothly with C modules. However, it is ultimately the application programmer’s responsibility to provide methods to handle the various run-time exceptions that may be raised when calling external C modules.

To help you build an effective exception-handling strategy into your application, this chapter explains the DLL and C Connect exception mechanism in detail.

External Interface Exceptions

Broadly speaking, exceptions may be raised under four circumstances:

- When a C function call in your ExternalInterface class returns with an error.
- When the attempt to access a function or datum in an external library fails.
- When the attempt to access a datum on the external heap fails.
- During macro evaluation.

C function call failures

Exceptions in this category can occur as a part of the normal execution of your application. For example, a call to allocate a block of storage on the external heap will raise an exception if there is insufficient memory. Your application should attempt to recover gracefully from exceptions of this variety, performing any cleanup or recovery that may be necessary.

Library access exceptions

Exceptions that fall in this category tend to occur during system installation or code development when all the requisite external library files may not be properly in place. In this case, either the external C code module cannot be loaded, or else functions or static variables in the module have not been properly exported.

C datum exceptions

Exceptions that are raised when your application attempts to access a datum on the external heap may be caused by illegal type coercions (between Small- and LargeIntegers, for example), or by attempts to de-reference invalidated pointers to the heap (for example, after a snapshot).

C macro evaluation exceptions

The exception ExternalInterface >> errorSignal may be raised during C macro evaluation. Since DLL and C Connect currently does not support full macro evaluation, this signal indicates that you are trying to parse a type of #define statement whose semantics are not understood. Typically, this only occurs during the early stages of development when you are first parsing C header files to build your interface classes. For more details on working around these exceptions, see Syntax Errors.

The first three types of exception may occur during the run-time execution of your application, while the last one (macro evaluation) can only occur when you are first developing your interface classes. For this reason, your error handler need only treat the first three types of exception. In the following sections, we will discuss these three types of `ExternalInterface` exception in more detail.

C Function Failure

The first variety of exception you are likely to encounter arises during a C function call. As a rule, the Object Engine's calling mechanism does not require C function calls that return with an error to raise exceptions, but by convention methods in class `ExternalInterface` should send signals on failure. As discussed in the section on C function calling (see Calling C Functions), Smalltalk methods that contain function prototype declarations can also contain failure code. The failure code consists of Smalltalk expressions, and it is only invoked when the C function fails to return a value. The code can be any sequence of Smalltalk code, but by convention it follows a “standard” error handling protocol provided by class `ExternalInterface`.

This is illustrated in the following example method:

```
atol: aString
 <C: long atol(const char * )>
 ^self externalAccessFailedWith: _errorCode
```

By default, the method `ExternalInterface >> externalAccessFailedWith:` raises an `externalAccessFailedSignal.` In the code fragment shown above, the signal will be raised with the `SystemError` object that contains the exact cause of the failure.

The type of error is stored in the `name` and `parameter` fields of the `SystemError` object. These fields can be tested by the error handler.

The following table lists all the possible named errors that may be returned to the caller by this mechanism:

Table 20: SystemError identifiers

Name	Parameter	Details
#'allocation failed'	nil or number	Some allocation attempt of a Smalltalk object was made before making the C function call, and the object was needed to hold the result of the call. The allocation failed because of a shortage of Smalltalk memory. The number of bytes required for allocation may be returned in parameter.
#'bad argument'	index	An argument to the C function call was invalid. The parameter is the 1-relative index of the offending argument.
#'bad handle'	nil	The handle of an ExternalMethod was invalid. It should be an integer representing the address of a function.
#'C allocation failed'	nil or number	Some allocation attempt of C memory was made before making the C function call, and the memory was needed to hold the result of the call. The allocation failed because of a shortage of memory on the external heap. The number of bytes required for allocation may be returned in parameter.
#'exception occurred'	#(code pc)	An exception occurred during an external call. The parameter is an array of the platform dependent exception code and the program counter. On Windows platforms, the exception code is returned by the structured exception handling intrinsic function _exception_code. On UNIX platforms, the exception code is a signal value.
#'hresult error'	error code	The called function returned a value less than zero. The parameter is

Name	Parameter	Details
		that negative value. This is currently supported only on Windows platforms using COM connect.
#'io error'	error code	The called function returned an error according to the interpretation defined by one of the platform-defined calling conventions (discussed below). The parameter is the accompanying error code.
#'object engine internal error'	—	An internal object engine error occurred while trying to perform the call.
#'out of threads'	nil	A process attempted to perform a threaded external call, and a new thread was required but none could be allocated.
#'threaded api error'	error code	An internal error in the thread management system occurred. The error code identifies the nature of the problem which you should report to technical support.
#'unsupported operation'	nil	An operation unsupported on the current platform has been attempted (e.g., calling a `_wincall` error convention function call on a non-Windows platform).

Notice that there are two different sorts of errors that might be returned by C function calls: first, there are errors which are generated by the Object Engine and do not depend upon specific features of the platform in use; and second, there are errors that may contain some platform-specific information. Your error handling code should treat both of these types of error.

In the first group of errors (those that are not platform-specific), there is one common error which has complicated semantics that your error handling code should be aware of.

The error name `#'bad argument'` indicates that you passed an invalid argument to the C function. The parameter of `_errorCode` is the integer N, where N indicates which argument is invalid. N = 1 indicates the first (leftmost) argument. For example, the `atol()` procedure is typed to accept one character pointer argument and to return a long. If you passed a Smalltalk OrderedCollection object as the argument, your failure code would be invoked with _errorCode's parameter set to 1. If N = -1, this indicates that some sort of memory violation occurred, i.e., you are either de-referencing a NULL or invalid pointer, or writing over some zone of memory that you shouldn't be. The types of the arguments may be correct, but you should check the values of the arguments to make sure they are valid.

If you receive an externalAccessFailedSignal when calling a C function which uses the DLL and C Connect Object Engine Access interface (for details on this interface, see the chapter Object Engine Access Reference), you should refer to the Appendix section Object Engine Access Interface Exceptions.

The second group of errors (i.e., those that are platform-specific) includes #'io error' and #'hresult error'. The semantics of these errors are described in the following table:

Table 21: SystemError identifiers

Name	Platform	Details of Error Convention
#'hresult error'	Windows Only	This is the COM Connect error convention. If the called function returned a value less than zero, the parameter is set to that negative value. Indicated by the _hresult or __hresult initializer.
#'io error'	Unix	This is the UNIX error convention. If the called function returns -1, then the parameter is set to the value of errno immediately after the called function returned. Indicated by the _syscall or __syscall initializer.
#'io error'	OS/2	This is the Windows API error convention. If the called function returns zero, then the parameter is set to is the value returned by calling WinGetLastError() immediately after the called function returned. Indicated by the _wincall or __wincall initializer.
#'io error'	Windows	This error may be returned by Windows, according to either the _syscall or the _wincall convention. If the called function returns a negative value, then the _syscall convention is being used and the value of errno is negative.

Name	Platform	Details of Error Convention
		If the called function returns a positive value, then the _wincall convention is being used and the error code value is positive. Regardless of the convention being used, the parameter is set to the error code. If the _wincall convention is being used, the value is set to the result of GetLastError(), while if the _syscall convention is being used, the value is set to the result of errno().

Additional platform-specific details about decoding #'io error' parameters can be found in the section Exception Error Codes.

Note that if you do not specify failure code in your Smalltalk interface methods, the default is to answer the receiver. Obviously, this can produce unpredictable results. You should strongly consider adding failure code to any external method with which you need to detect failure conditions. To simplify the implementation, the best way to code these signal handlers is to write “wrapper methods” in your ExternalInterface class.

External Library Access Exceptions

In addition to the ExternalInterface exceptions described in the preceding section, there is a second category of errors that can be generated by the DLL and C Connect calling mechanism. These errors may arise because of difficulty accessing functions or static variables in external libraries. Before calling a function or accessing an external variable, the system must first locate and load the corresponding DLL file. If these library file(s) cannot be located or loaded, DLL and C Connect will raise an exception. Since these errors occur before functions or data can be accessed, they tend to arise during development or customer system installation. For details on locating and loading libraries, see Dynamic-Link Libraries.

There are three signals that can be raised either upon the failure of a call to a C function, or upon failure when accessing a datum contained in an external library. One additional signal, libraryNotUnloadedSignal, may be raised during the attempt to unload libraries from the run-time system.

The following table summarizes the significance of these four signals:

Table 22: External Library Access Exceptions

Signal:	Description:
libraryNotFoundSignal	This exception indicates a failure during the attempt to locate a library specified in the ExternalInterface's class creation template. The library could not be located on the specified path.
libraryNotLoadedSignal	This exception is raised for a failure during the attempt to load the library that your ExternalInterface class indicates contains the target function or datum. When you first attempt to access a function or datum, the ExternalInterface class must load the corresponding library file.
libraryNotUnloadedSignal	This indicates a failure during the attempt to unload a library specified by the libraryFiles attribute in your ExternalInterface class.
externalObjectNotFoundSignal	This exception indicates that the product successfully loaded the library associated with your interface class, but subsequently could not find the external object either within the library or as a statically linked object. This exception typically indicates a symbol-lookup failure in the external module.

To catch these signals, you can write signal handlers in the callers of your interface methods. To simplify the code, the best way to do implement these signal handlers is to write wrapper methods in your ExternalInterface class.

C Datum Access Exceptions

In addition to the exceptions raised upon failure of C function calls and external library references, there is a third type of exception that can occur when your application manipulates C data objects on the external heap.

The following table summarizes the significance of these signals:

Table 23: C Data Class Exceptions

Signal:	Description:
illegalAssignmentSignal	This exception indicates that the argument to a method which performs an assignment operation is invalid with respect to the receiver’s referent type.
memberNotFoundSignal	This exception is raised if the argument to memberAt: or memberAt:put: does not represent a valid member name for an instance of class CCompoundType.
invalidNumberOfArgumentsSignal	This exception indicates that the wrong number of arguments were passed to a C function.

Chapter

7

Packaging Considerations

Topics

This chapter explains how to prepare your C interface classes for delivery in such a way that your deployed image does not rely on the development tools that come with DLL and C Connect. This chapter also demonstrates how to relink libraries on the customer's machines.

General Considerations

After designing and building your external interfaces you will have a collection of ExternalInterface subclasses. The source code embedded in the methods of these classes contains special syntax that can only be parsed by the C declaration parser. To avoid this dependency, you can use parcels to unload and reload your interface classes. Because parcels are provided in the base VisualWorks product, DLL and C Connect does not need to be included with your distribution to run your application.

Note: In VisualWorks 3.0, the Binary Object Streaming Service (BOSS) was supplanted with the parceling framework. Although BOSS may still be used to unload and reload your interface classes, it is strongly recommended that you use parcels instead. For a more detailed discussion of parcels, see the VisualWorks *Application Developer's Guide*.

A second consideration with packaging involves the C macro definitions that are compiled into instances of the class CMacroDefinition. As a component of DLL and C Connect, this class is not part of the base VisualWorks image. However, DLL and C Connect provides protocol that will convert each instance of a CMacroDefinition into the scalar Smalltalk value it represents (Number or String).

A third consideration involves the option to compile external interface function accessing methods in a fully optimized form. Compiling these methods in an optimized form allows them to run much faster at the expense of secure argument type checking.

Preparing Your Interface Classes

The procedure for distributing interface classes involves two steps:

1. Recompiling the Interfaces in optimized form.
2. Resolving references to class CMacroDefinition.

When you have completed your debugging cycle and wish to distribute your interface classes, the first step is to recompile them in an optimized form. This form takes less memory and allows the interfaces to run much faster, at the expense of secure argument type checking. You can accomplish this in one of two ways:

- Modify the external interface class definition template by changing the argument to the optimizationLevel: keyword from #debug to #full, and then accept the change. This will recompile every method in the class using an optimized calling sequence.
- Send the message optimizationLevel: to your interface class with #full as the argument. This will also recompile every method in the class with an optimized calling sequence.

Note: After you have compiled your interface class in an optimized form, you no longer have access to the ExternalProcedure objects that are contained in the interface's pool dictionary. These objects contain type information that is no longer needed in a run-time environment.

The second step in preparing your interface class is to remove any references to CMacroDefintions. Methods that contain C #define statements compile into instances of CMacroDefinition. However, this class is not available in a run-time environment. It is provided only as a development time tool that enables you to dynamically evaluate a macro's definition. Once your interface class is ready for distribution, you can replace every instance of a CMacroDefinition with its corresponding scalar value. To do this, you send the message fillDefineCachesWithValues to each of your interface classes.

To summarize, each interface class in your project should be sent the following methods before packaging:

```
TheInterfaceClass optimizationLevel: #full.
TheInterfaceClass fillDefineCachesWithValues.
```

Packaging Your Interface Classes

Parcels are now the preferred means of packaging and exporting External Interface classes. You may parcel out your External Interface classes either using the External Interface Finder, or using a special protocol provided by class ExternalInterface.

To parcel out the example class StandardLibInterface using the Finder tool, perform the following steps:

1. Within the Finder tool, select class StandardLibInterface in the **Class** list.
2. Choose the Class > Parcel Out As... menu option to select parcelling options and actually write the parcel file.

Normally, you will use the default options selected in the dialog box.

The ExternalInterface class also provides a set of utility methods for parceling itself (or, more likely, a subclass of itself) out. For example, the following code fragment would create a parcel file named stdlib.pcl containing the class definition and methods for the example interface class StandardLibInterface:

```
| parcel |
 parcel := Parcel name: 'stdlib.pcl'.
 StandardLibInterface
  parcelClasses: StandardLibInterface withAllSubclasses
  toParcel: parcel.
 parcel saveParcelDialogFor: nil
```

This code fragment will open a dialog box to prompt for parcelling options. You should use the default options.

You can manage your parcel files using a Parcel Browser; for details, see the VisualWorks *Application Developer's Guide*.

Relinking C Libraries

The C libraries on which your application depends may be located in a different location in the deployment environment. It is the responsibility of your installation procedure to determine the proper pathnames. The ExternalInterface class provides a utility protocol for re-establishing the linkages, as in the following examples:

```
"When the stdlib library is named newStdLib in the user's environment..."
StandardLibInterface libraryFiles: 'newStdLib.dll'
"When the stdlib library is located in a different directory..."
StandardLibInterface libraryDirectories: 'c:\windows\system'
```

Each of the path-resetting messages shown above first unloads the libraries so they will be relinked the next time they are invoked. You can perform the unloading part of the operation separately:

```
StandardLibInterface unloadLibraries
```

You can also make use of environment variables to specify library paths that will be valid across a number of different platforms. For details, see Dynamic-Link Libraries, “Libraries and Environment Variables.”

Chapter

8

Platform-Specific Information

Topics

This chapter provides an overview of platform-specific development, followed by discussions of two issues. The first issue concerns the platform independent requirements for registering entry-points to code modules that have been statically linked into the Object Engine. The second issue concerns the specifics of developing applications for particular platforms.

The bulk of this chapter discusses the different supported platforms, outlining the constraints imposed by the idiosyncrasies of the various platforms. Each section details the platform specific information for using DLL and C Connect with dynamic-link libraries as well as statically linked Object Engines.

Platform-Specific Development

VisualWorks is distributed on a variety of platforms. Most of the files contained in the distribution are platform independent. However, some of the files are only relevant to a particular release platform of VisualWorks. All platform dependent files are located in separate subdirectories in the distribution directory tree. All other files are the same across platforms, except for the line-end conventions used in text files.

A DLLCC test suite shared library files can be found in each appropriate platform subdirectory provided with the product. You can use the test libraries to verify that the product is operating correctly before you begin using it with your applications. The test libraries can be complied and linked with the contents of the `dllcc/src` directory makefiles that are provided in each subdirectory. For example:

```
dllcc/platform_directory/makefile
```

Compiler Compatibility

The following table lists some of the more popular compilers known to work with VisualWorks 8.3. This is not a complete list and is intended only as a guide to developers selecting a compiler to build external code components. As a general rule, VisualWorks 8.3 is usable with any ANSI C compatible compiler conforming to the platform's language calling conventions.

Table 24: Compatible Compilers

Platform / OS Version	Compiler
Linux 32-bit on x86	Minimum: kernel 2.4, glibc 2.2.4, gcc 2.96 (e.g. RedHat 7.2)
Linux 64-bit on x86	Minimum: kernel 2.4, glibc 2.3.2, gcc 3.2.3 (e.g. RedHat Enterprise Linux ES 3)
Linux 32-bit on PPC / POWER	Minimum: kernel 2.6, glibc 2.5, gcc 4.1.2 (e.g RedHat Enterprise Linux Server 5.4)
IBM AIX 6.1	IBM XL C/C++ for AIX, version: 10.1.0.11
Mac OS X	OS X 10.4 or newer, Xcode gcc

Platform / OS Version	Compiler
MS-Windows 32-bit XP (SP3) and later	MS 32-bit C/C++ Optimizing Compiler Version 16.00 or later
MS-Windows 64-bit	MS 64-bit C/C++ Optimizing Compiler Version 16.00 or later
Solaris 8, 9, or 10 on SPARC	Sun WorkShop 6 update 2 C 5.3 2001/05/15
Solaris 10 on x86	Sun Studio 10 or later

Unsafe Compiling

DLL and C Connect performs type checking on object-oriented pointers (`oops`) automatically. However, such runtime checks may not be desirable when performance is your primary concern.

For situations in which critical performance needs motivate you to bypass the safety mechanism, you can compile your code defining the symbol `UNSAFE`. Note that this will have no impact if your code does not handle object pointers.

Caution: We do not recommend that you use `UNSAFE` in ordinary practice.

Even in “safe” mode, your C functions should perform consistency checks on passed arguments — at least checking for the correct class. When you use the `UNSAFE` implementation, doing these checks is absolutely essential. Test your C code *very* carefully before compiling `UNSAFE` and linking the result to a working image.

Some platforms do not support “back-referencing” from a dynamic-link library into the Object Engine. For this reason, these platforms cannot compile code that uses the Object Access functions in `UNSAFE` mode — they must use the safe, functional, version.

Caution: If you compile your C code with `UNSAFE`, problems in your library can cause the Object Engine to fail, or even corrupt the virtual image. Because hard-to-trace errors can be introduced so easily in unsafe mode, trouble calls relating to an Object Engine that is linked to unsafe code will not be handled by Cincom technical support.

Incremental Loading of Dynamic-Link Libraries

Many platforms support incremental loading of dynamic-link libraries. This can occasionally cause problems during execution if several libraries contain interdependencies. For example, if a function in library Q is dependent on a global variable in library P, the symbol lookup in P can fail unless both Q and P are loaded. The failure will raise an `externalObjectNotFoundSignal` or a `libraryNotLoadedSignal` exception. Worse, if a function in library Q depends on a function in library P, on some platforms execution can be halted due to the unresolved symbol.

The solution to this problem is to make sure that all libraries that contain interdependencies are specified in the `libraryFiles` attribute of your `ExternalInterface` class. This will effectively force them to be loaded together. The `libraryFiles:` protocol provides a means for ensuring that a group of libraries can be loaded together. It is important to note, however, that the order in which you specify the libraries is important. If you use a library R which in turn depends upon library Q which itself depends upon a third library P, then the `libraryFiles` attribute of your `ExternalInterface` class must have P specified before Q which must be specified before R.

Static Linking

DLL and C Connect provides access to external code (code outside the domain of the Smalltalk object memory) either in the form of dynamic-link libraries or libraries statically linked to the run-time system. This section describes what needs to be done when building a statically linked Object Engine to register the external entry-points that are referenced by your `ExternalInterface` classes. VisualWorks in general favors interfaces designed to use dynamically-linked modules.

For those who are familiar with building an Object Engine, the process of creating a custom Object Engine with DLL and C Connect is similar. The procedure relies on the use of a platform dependent makefile that is described later in this section. The steps are as follows:

1. The Object Engine maintains a registry of external objects. You must arrange for an entry to be created in this registry for each of your externals. To do so, create an `oeInitLinkRegistry()` function that makes an `oeRegisterSymbolAndHandle()` call for each exported symbol. The following example shows how to register four functions from the **`string.h`** library, as well as a global variable:

   ```
   void oeInitLinkRegistry(void)
   {
    oeRegisterSymbolAndHandle("strcmp", strcmp);
    oeRegisterSymbolAndHandle("strcpy", strcpy);
    oeRegisterSymbolAndHandle("strlen", strlen);
    oeRegisterSymbolAndHandle("strstr", strstr);
    oeRegisterSymbolAndHandle("errno", errno);
   }
   ```

   The first argument to the registering function `oeRegisterSymbolAndHandle()` is a string containing the name that is invoked by the Smalltalk method; the second argument is the name of the C function. By convention, these two are the same. For example, the string comparing method that invokes the strcmp would look like the following:

   ```
   strcmp: string1 with: string2
    <C: char *strcmp(const char *string1, const char *string2)>
   ```

2. A platform-specific makefile is provided in the **`make`** subdirectory. It links a test file (**`cpoktst.c`**) that defines and installs a set of test functions. Edit **`cpoktst.c`** (also found in the **`make`** directory), adding code for your custom functions and, if you desire, remove the code for the test functions. For those who are familiar with user-defined primitives, this file is analogous to **`validate.c`**. The makefile also refers to **`validate.c`** — in that file, remove the test code, leaving the following external entry point for user primitives (expected by the Object Engine):

   ```
   char *
   oeInstall(void)
   {
   }
   ```

3. In the makefile, you may also need to identify the location of any libraries that you use.
4. Run the makefile. By default, the custom Object Engine is named `VisualWorksUser`, though you can edit the makefile to give the target file a different name. The `VisualWorksUser` file is your new Object Engine, to be substituted for the executable installed in your system.

Mac OS X

DLL and C Connect supports dynamic loading of Mac OS X libraries at runtime. Various types of dynamic libraries are available, each with distinctive properties. Dynamic libraries may be individual files, or they may be packaged within "bundles". Using DLL and C Connect, you may access either type of library.

Dynamic Libraries

Shared libraries (files with the extension `.dylib` or `.so`) may be dynamically loaded into OS X applications at runtime. The command to load the library may be implicit or explicit. The implicit load occurs as a consequence of ’statically’ linking an executable with the library. OS X handles the loading of the library automatically at runtime, resolving all symbol references.

DLL and C Connect may be used to explicitly load a `.dylib` library, but there is no mechanism to unload it (because it will appear to have been statically linked to the object engine). If you wish to both load and unload libraries from a running executable, you must use bundles.

Bundles

Mac OS X delivers applications, frameworks, and loadable libraries as bundles. Bundles are essentially directories used to group executable code and runtime resources into a single entity. From the user’s perspective, the bundle appears to be a single unit. DLL and C Connect provides support for handling two of the three varieties of OS X bundles:

Frameworks

Bundles with a `.framework` extension contain dynamic shared libraries and associated resources (e.g. header files, images, and even documentation). Typically, these are packaged Objective-C class libraries (roughly corresponding to VisualWorks parcels).

Loadable Bundles

Entities with a `.bundle` extension are similar to frameworks and applications,but they are designed to be explicitly loaded into a running application (i.e., application plugins).

For additional details on OS X bundles, see:

```
http://developer.apple.com/documentation/MacOSX/Conceptual/ SystemOverview/
Bundles/chapter_5_section_1.html
```

MS-Windows

VisualWorks DLL and C Connect runs within Microsoft’s Win32s subsystem. This subsystem requires all dynamic-link libraries that run under Win32s to use this 32-bit flat memory model.

Details about using 32-bit Dynamic-Link Libraries are discussed in the following section on Windows XP. Access to 16-bit DLLs is no longer supported.

When static-linking an Object Engine for a Windows platform, it is necessary to define the CPU environment variable as follows:

```
CPU=i386
```

Object Engine Access Interface with MS-Windows

To access the Object Engine interface from a statically-linked DLL, you should use the `vwnt.exe` rather than the `visual.exe` engine. The `visual.exe` engine does not use the `vwntoe.dll` required for Object Engine access, and `.exe` files do not export their symbols.

To access this interface, you will need to perform the following steps to ensure that your project links correctly. If you are using the Microsoft Visual C++ compiler, normally you would export the symbol `oeLoadInitialize()` by adding the following line of code to your C/ C++ file:

```
extern __dllspec(dllexport) oeLoadInitialize(void);
```

However, this does not work due to the nature of the `dllexport` attribute and the fact that the `oeLoadInitialize()` function does not have a definition in any of the dependent modules. What you must do is actually create a `.def` file and add the symbol to `oeLoadInitialize` in its `EXPORTS` section, making sure to insert the `.def` file into the project. For details, see the Microsoft documentation for information on how the loading of exported symbols from a **`.def`** file differs from the loading of symbols from a **`.cpp`** file using the **`dllexport`** attribute.

MS-Windows XP and Vista

32-bit Dynamic-Link Libraries

DLL and C Connect supports the Microsoft Visual C/C++ compiler. Two standard calling conventions are supported:

`__cdecl`

Arguments are pushed on the stack, in reverse order (right to left). The caller pops arguments.

`__stdcall`

Arguments are pushed on the stack, in reverse order (right to left). The callee pops arguments.

Callbacks can only use the `__stdcall` calling convention. You must be very careful to tag your callback pointer types with the appropriate calling convention or unpredictable results will occur (typically stack corruption followed by a memory exception).

Consult the MS-Windows Visual Studio documentation for further information on how to build dynamic-link libraries.

Dynamic-link libraries that use the Object Engine access protocol (for further details, you may consult the chapter Object Engine Access Functions) must export the symbol `oeLoadInitialize`. This function is called when the library is first loaded into the address space of the calling process to initialize local data used to implement the Object Engine access protocol. This symbol should appear in the EXPORTS field of your module definition (.DEF) file.

Using 32-bit DLLs on 64-bit MS-Windows

When using 32-bit DLLs on a 64-bit version of MS-Windows, note that 32-bit DLLs should be placed in the **SysWOW64** directory. 64-bit DLLs should reside in the **System32** directory.

Structure Layout Issues under MS-Windows XP and Vista

When working with Windows XP and Vista include files, it is sometimes necessary to accommodate unusual byte-packing algorithms used for structure layout. Some include files, through the use of a #pragma compiler/preprocessor directive, can modify the layout of structures. Because VisualWorks ignores all compiler-specific #pragma directives, this can lead to incorrect execution of your interface classes. A specific example is the Windows XP header file **commdlg.h**, which defines the following #pragma and structure declaration:

```
#ifndef RC_INVOKED
#pragma pack(1) /* Assume byte packing throughout */
#endif
typedef struct tagOFNA {
 DWORD   lStructSize;
 HWND    hwndOwner;
 ...
 LPCSTR   lpTemplateName;
} OPENFILENAMEA;
...
```

You can specify a a byte packing algorithm for this structure by realigning the structure during image start-up. In your interface class's #installOn: method, execute the following expression:

```
YourInterfaceInstance OPENFILENAMEA
 typeDo: CStructureLayout dosLayout
```

You must perform this realignment for every structure that uses byte packing versus the standard 32-bit packing algorithm. For a longer discussion of alignment algorithms, see External Heap Alignment.

Declaring the C Functions

When defining the interface for a DLL, it is important to know the calling conventions used by the external function entry points in the DLL. This information is usually available in the DLL's interface manual or the C language header files associated with the DLL.

Two calling conventions for Windows DLLs are supported. The `__stdcall` is used to call Win32 API functions. The `__cdecl` convention is the default calling convention for C and C++ programs. Refer to the Microsoft documentation for more information about using these conventions.

An example of a method to call a Windows C function is as follows:

```
cProcedure
 <C: long __cdecl cProcedure(long arg1)>
```

Ordinals

Procedure addresses within a DLL are obtained by using the symbolic name of the entry point or by specifying the ordinal number associated with the entry point. The ordinal number is an integral number unique to the entry point. Ordinal numbers allow address look-up to be accomplished much faster than using a symbolic name. Ordinal numbers are assigned by the DLL designer, and they can be obtained by using the **exehdr** application that is typically bundled with your compiler.

An ordinal number can be associated with a function. If you specify a function's ordinal number, that number is used during address look-up. If not, the function's name is used.

To associate an ordinal number with a function, send the message `ordinal:` to the external procedure object that represents the function if your interface is compiled in `#debug` mode. The external procedure object is obtained from the pool dictionary associated with the interface class. You can use the name of the function if your Smalltalk method has access to the pool dictionary.

For example, given the following interface class method:

```
aFunction
 <C: long aFunction(void)>
```

Assign the function an ordinal number using the following statement:

```
aFunction ordinal: anInteger.
```

Note that this mechanism will not work correctly if the interface is compiled fully optimized. In this case, you must send the `ordinal:` message to the `ExternalMethod` object that is created when a function prototype method is compiled. In the above example, you would need to evaluate the following:

```
| compiledMethod |
compiledMethod := SomeInterface compiledMethodAt: #aFunction.
compiledMethod ordinal: anInteger.
```

Declaring the C Data Types

When writing interface code that defines C data types for Windows DLLs, observe the following conventions:

Default datatype sizes:

The default size for types varies among the platforms supported. Be careful about defining portable C data types if you want your code to be usable across platforms.

Structure order, alignment and padding:

The layout of structure members, their alignment within the structure, and the padding between members is implementation dependent. The default Windows structure alignment algorithm when running on a Windows platform is used, which packs structure members on one byte boundaries. You can modify the layout by adding a new default layout to the `CStructureLayout` class. This, however, is not necessary unless you are calling a DLL that uses a different structure layout.

Byte order within a word:

The 80x86 processor line stores words in memory with the most significant byte last. Each byte is stored with the most significant bit first. For portability, your code should not rely on any word byte ordering.

Bit fields:

Bit field structure members are not supported.

Type specifiers:

The `long double` type specifier is also not supported. For the Microsoft compilers it defines an 80-bit floating-point quantity.

Strings and string formats:

It is important to know the format of strings when you must pass string pointers to the DLL. For more details, refer to the sub-section Strings.

Pointer arithmetic:

All pointer arithmetic is performed on 32-bit pointers. Note that the `_huge` type qualifier is not supported by DLL and C Connect.

Strings

Smalltalk `String` objects are specially treated. They can be passed as function arguments typed as `char *`.

In the case of `String` arguments, a direct pointer to the Smalltalk string is passed if the string can be properly `NULL` terminated. If not, Smalltalk makes a `NULL` terminated copy of the string in its object memory and a pointer to the new copy is passed as the function's argument.

Since it is possible that Smalltalk copied the `String` argument, do not assume that Strings can be destructively modified. If you wish to destructively modify a string, you must first copy the string to the heap and then copy it back into a Smalltalk string object after the function returns.

Various languages implement strings in different ways. If you are calling a C routine that accepts a language specific string as an argument of type `char *`, you must make sure to pass a pointer object that implements the particular string format. For example, Microsoft's Pascal format stores strings as a length byte followed by string data, or as a fixed length sequence of `char`s. Microsoft's BASIC format is to store strings as a four-byte descriptor. The first two bytes store the string's length while the second two bytes are the offset of the string data relative to the current data segment. Microsoft's FORTRAN format is to store strings as a fixed-length sequence of characters. As stated in the Microsoft documentation, variable length FORTRAN strings cannot be used in mixed-language programming because the hidden variable used for string length is not accessible.

Callbacks

Only callbacks that conform to the `__stdcall` calling convention are supported. An example of a legal C language type declaration of a callback pointer is as follows:

```
typedef long (__stdcall *CallbackAddPtr)(long, long);
```

`CallbackAddPtr` is a function pointer type defining a function that accepts two `long` arguments and returns a `long`.

The following is a small example of the declarations needed to define and use a callback from a Windows DLL. The first step is to define the callback pointer type that is used in the function declaration and is also used in generating the `CCallback` object. This is performed by defining the following type declaration in your interface class:

```
CallbackAddPtr
<C: typedef long (__stdcall *CallbackAddPtr)(long, long)>
```

The second step is to define the function entry point in the DLL. This is accomplished by entering the following Smalltalk method into your interface class:

```
add: functionPtr with: arg1 with: arg2
 <C: long _cdecl add(CallbackAddPtr, arg1, arg2)>
```

Finally, define a method that creates the Smalltalk callback object and invokes the DLL function as follows:

```
testCallback
 | callback |
 callback := CCallback do: [:arg1 :arg2 | arg1 + arg2]
  ofType: self CallbackAddPtr.
 ^self add: callback with: 1 with: 2
```

An example implementation of the `add()` C function that is compiled into the DLL is:

```
typedef long (__stdcall *CallbackAddPtr)(long, long);
ong __cdecl
=add(CallbackAddPtr callback, long arg1, long arg2)
{
 return (*CallbackAddPtr)(arg1, arg2);
}
```

Library Search Paths

When DLL and C Connect needs to search for a C library to load, the library names and paths specified in your library's Smalltalk interface class are used. However, if your interface class does not specify a complete path to the library, the same searching mechanism is used as is used by the Windows API `LoadLibrary()` function. This mechanism searches for the library in the following locations:

1. The current directory.
2. The Windows directory (the directory that contains `WIN.COM`).
3. The Windows system directory (the directory containing system files such as **`GDI.EXE`**).
4. The directory containing Smalltalk's executable file.
5. The directories listed in your **`PATH`** environment variable.
6. The list of directories mapped in a network.
7. If the library is not found in any of these locations, Smalltalk raises an exception (a notifier may or may not appear on your screen, depending on the exceptions handlers that wrap the call). Note that you can make use of environment variables to

simplify the path specified in your interface class (for details, see Dynamic-Link Libraries).

Defining the DLL Interface

This section describes the basics of writing and compiling your own dynamic-link libraries. It is intended as an introduction to the concepts of building a DLL from the C programmer's viewpoint, rather than that of a Smalltalk programmer. It is by no means a complete description of the nuances and intricacies of writing DLLs. Rather, it describes the salient features that you should consider when designing and writing a DLL.

There are four aspects to writing a DLL:

- The external API must be defined.
- The DLL definition file must be created.
- The API code must created and compiled.
- The DLL must be created.

The first step in creating your own DLL is to properly define the interface. It is important that you plan the interface well, to provide a full set of services and to minimize maintenance costs. Important points to remember when defining the interface are:

- Declare all function entry points either `__cdecl` or `__stdcall`.
- Declare all function entry points `_export` if the function's name does not appear in the `EXPORTS` section of a module-definition file. For more information on defining exports, consult Microsoft's module-definition file documentation.
- Be aware of the separate data and stack segments that exist when your code enters DLL functions. You may need to declare DLL function entry points with the `_loadds` type qualifier. The `_loadds` qualifier causes the compiler to add entry and exit code to the function. The entry code loads the `DS` register. Loading the `DS` register adds some overhead, so limit this to only exported functions. Note that you can use compiler memory model options to control segment setup. See your compiler documentation for more details.

Creating the Definition File

The second step in creating your DLL is to create the module-definition file. This section outlines only the minimum requirements for generating a Windows DLL module-definition file. Consult your development environment's documentation on building module-definition files for more detailed information.

The following entries should appear in the module-definition file:

- The LIBRARY statement. This statement identifies the file as a DLL.
- The DESCRIPTION statement (optional). This statement inserts text into the DLL and is used to insert copyright strings and/or more fully describe the purpose or version of the DLL.
- The EXETYPE statement. This statement indicates which operating system the DLL was created for. Its value will be WINDOWS [[version]].
- The EXPORTS statement. This statement defines the names of the DLL's exported functions. Note that some language compilers allow exporting function names by using the _export type qualifier.

There are other module-definition statements that are relevant to building DLLs. Some of the more useful statements are HEAPSIZE, CODE, DATA, SEGMENTS, and IMPORTS. Refer to the appropriate manual for a detailed description.

An example definition file that exports two functions appears below. The first function, Cadd(), is a C style function. The second function, PascalAdd(), is a Pascal style function. The next section declares the actual functions.

```
LIBRARY    TESTLIB
DESCRIPTION     'Test Library Version 1.0'
EXETYPE    WINDOWS
EXPORTS
 Cadd = _Cadd
 PascalAdd
```

Compiling the External Library Code

The third step in creating a DLL is to compile the DLL's source code. There are many compiler options for the various vendor's compilers. Consider the following four items during the compilation process.

Compile Without Linking

You can compile each source file into an object file incrementally rather than all at once. This can be controlled by using the development environment's makefile facility. For the Microsoft compilers, use the `/c` switch.

Large Memory Model with Separate Stack

You must make sure to specify the correct memory model for your DLL code. A typical model is the large memory model. In addition to the memory model, you must indicate segment setup. For DLLs, because there is a separate stack and data segment, you should specify if the `SS` and `DS` registers get loaded during function entry. Refer to the specific compiler documentation for the appropriate flags.

Specify Code for the Appropriate Processor

You must specify what type of machine code the compiler generates (for '86, '286, '386, '486, or Pentium architectures). Refer to the specific compiler documentation for the appropriate flags.

Packed structures

You must specify which structure packing layout will be used. It defaults to using the packed structure layout algorithm. For the Microsoft compiler, use the `/Zp1` option.

For example, assume the hypothetical file **`SAMPLE.C`** contains three functions. The following commands, based on the Microsoft C/C++ 7.0 compiler, will compile the file into a DLL. You need to substitute equivalent commands for the compiler on your system.

```
#include <windows.h>
int CALLBACK
LibMain(
 HINSTANCE hInstance,
 WORD  wDataSeg,
 WORD  cbHeap,
 LPSTR  ignore)
{
 return 1;
}
long _cdecl
Cadd(long a, long b)
{
 return a + b;
}
long _stdcall
PascalAdd(long a, long b)
{
 return a + b;
}
```

To compile SAMPLE.C, execute the following command from a DOS prompt. This assumes that you have correctly initialized your PATH and LIB environment variables.

```
CL /c /ALw /G2 /Zp SAMPLE.C
```

This generates the file SAMPLE.OBJ, which is linked and converted into a DLL. Remember that the compiler needs to be able to access the WINDOWS.H file included by the sample program. This include file is found in the developer's kit. The next section describes the linking step.

Creating the DLL

Once you have compiled your DLL's source code as described in the previous section, you need to link the resulting object files and create the final DLL. There are various ways of linking the resulting object files, depending on the floating-point math requirements, and also depending on whether any run-time libraries must be linked. Consult your development environment's documentation for further library requirements.

In this example, there is one object file. To link the object file and create the DLL, execute the following command:

```
LINK /NOD SAMPLE.OBJ, SAMPLE.DLL, NUL, LDLLCEW LIBW,
SAMPLE.DEF
```

This command creates the dynamic-link library `SAMPLE.DLL`. Note that you need to link with two libraries. The first library, `LDLLCEW.LIB` is bundled with the Microsoft compiler. The second library, `LIBW.LIB`, is bundled with the Microsoft Software Development Kit (SDK). Consult the documentation for the compiler and the SDK for more information on these libraries.

A single link command option, `/NOD`, was specified. This option tells LINK not to search default libraries named in the object files.

The last step is to run the Resource Compiler on the DLL. Execute the following:

```
RC SAMPLE.DLL
```

This identifies a Windows version number with the DLL. You now have a DLL that can be loaded and accessed by your interface class.

Creating a Makefile

For general guidelines, consult your development environment's documentation on how to create a makefile.

Chapter

9

Object Engine Access Functions

Topics

DLL and C Connect was designed to interface Smalltalk with software written in C in a manner that makes C functions and data objects accessible and modifiable by Smalltalk. This chapter describes the Object Engine access functions, an analogous interface that provides a way for software written in the C programming language to access data objects and other functionality that reside in the Smalltalk Object Engine. These access functions replace the previous version of the interface that was known as the “User-defined Primitive” interface.

Overview

From the Smalltalk programmer’s point of view, DLL and C Connect provides two basic interfaces:

- A callout interface for calling C functions from Smalltalk.
- A callback interface for allowing called C functions to execute Smalltalk code.

As discussed in the chapter Calling Smalltalk From C, callbacks can be either (a) Smalltalk block closures that are called from a C function, or (b) external message-sends on Smalltalk objects that are performed via a set of C function calls. The functions that facilitate external message-sends are actually part of a larger collection of C functions known collectively as the “Object Engine Access Function” interface. This interface allows C code that you call from Smalltalk more flexible access to the Smalltalk object memory. With the exception of external message-sends, the Smalltalk Object Engine will not run during most of these access function calls.

Basic Capabilities

The Object Engine access function interface provides your code with a variety of capabilities and routines, including:

- Accessing Smalltalk objects.
- Following field references.
- Converting representations between C and Smalltalk.
- Creating new objects.
- Returning an object to a C function via a callback or message send.
- Indicating to Smalltalk that a C function callout did not successfully complete.

Predefined C Data Types

Seven C data types are predefined for the Object Engine Access interface. They are used to convert between Smalltalk objects and standard C data types according to the following table:

Table 25: C data types

C type	Converts to	C equivalent	Comment
OEoop			Reference to a Smalltalk object
OEbool	true/false	OETRUE/OEFALSE	
OEbyte	byte-type	unsigned char	
OEint	SmallInteger	long	
OEfloat	Float	float	
OEdouble	Double	double	
OEchar	Character	char	

Any object can be passed to, or received from, a function typed as OEoop.

The ExternalInterface class defines a new C type specifier _oop. This specifier indicates that a function argument or return value can accept an arbitrary Smalltalk object. A reference to the Smalltalk object is passed to the C function. To manipulate any object typed as _oop, use one of the special functions declared in the Object Engine access interface. The type specifier _oop used in your Smalltalk interface methods is equivalent to the C type OEoop.

Failure Codes

If an Object Engine access call succeeds, it returns a value to the caller. In case of failure, a special constant is available to indicate the failure condition. You may consult the individual function reference (see following pages) for a description of its failure conditions. If a function indicates failure, further information about the failure can be obtained by calling the function OEgetErrorCode(). This function returns an OEerrorCodeID, as defined in the file oeAPI.h (this file is located in the src subdirectory of the DLL and C Connect release). The possible results of this function are described in the following table:

Table 26: OEgetErrorCode results

Name	Description
OEerrorNone	No failure code available.
OEerrorCrange	A C argument is out of range; either because an index into a variable-length array is out of bounds, or else a Smalltalk datum is too big to be represented in C.
OEerrorNonOop	Oop argument not an oop.
OEerrorWrongClass	Oop argument incorrect type.
OEerrorObjectTooSmall	Smalltalk datum too small.
OEerrorAllocationFailed	Smalltalk object memory allocation failed.
OEerrorStoreFailure	The function illegally attempted to place an object reference into another object. This indicates an overflow of the “Remembered Table”; see the comments on class ObjectMemory for details.

Dynamic-Link Libraries

Dynamic-link libraries that use the Object Engine access protocol must export the symbol `oeLoadInitialize` (for further details, consult the chapter Platform Specific Information).

General Advice

- View `OEoop` objects as opaque objects, not actual object pointers (`OOPs`). The most important role of these handles is to be passed as object references to the Object Engine access interface.
- Your functions are capable of corrupting the Smalltalk object memory. It is especially important to perform bounds checking.
- Many operating system calls, such as `read()`, can cause Smalltalk to block until the call is completed. The end user may find it annoying if this system call cannot be satisfied within a reasonably short amount of time. As a rule, I/O operations should be performed using threaded calls (for details see the chapter Threaded Interconnect).
- There is no way to interrupt an Object Engine access or system call from Smalltalk. Typing control-Y does not work until the C function returns control to Smalltalk.

- The call stack above the current call frame does not conform to C calling conventions.
- Because memory allocations can fail, perform all allocations before provoking side effects in any data structure.
- An OEoop, Object Engine Access object pointer, is only valid during the current call to your function. If you try to save it across function calls you may cause function failure or a program crash because the object may be relocated by the Smalltalk memory manager. If an object is needed for more than one call, make it an instance variable of the receiver, pass it in as an argument for each call, or use the registry discussed in the section Registering Long Lived Objects.
- The C programming language uses 0-based indexes, while Smalltalk uses 1-based indexes.
- If a Smalltalk datum is too big to be represented in C, the Object Engine access routine fails and returns the OEerrorCrange failure code.

Registering Long Lived Objects

The Object Engine maintains a system registry of objects that must be referenced by Object Engine code. This is needed because the Object Engine relocates objects during memory management operations, so direct references to object memory (OEoop and _oop are such references) cannot persist across external calls to C functions.

To refer to objects over time, the Object Engine provides a facility to register indirect references to objects. Note that this registry is distinct from the object table that is used by the Smalltalk memory manager. These indirect references are indices in a table that the Object Engine memory manager keeps current.

Another important reason to use the registry is to reference an object that was given to an external function in a prior call. For example, imagine you have a function that was passed a Semaphore so it could be signalled later (by using oeSignalSemaphore()). Your function can ask the registry for a permanent slot (it returns a table index if there is enough room) and store the semaphore into the registry at that slot using the function oeRegisteredHandleAtPut(). You would then need to record the slot index in a static variable in the external code. Later, to obtain the reference the object, you can read the registry at the slot you recorded using the function oeRegisteredHandleAt().

Registry slots are a finite resource, and cannot be recycled (while the Object Engine is running), so use them sparingly. To reserve a registry slot, call:

```
static oeInt slot;
slot = oeAllocRegistrySlot();
```

Any slots you allocate (and the references you store in them) are discarded in a snapshot file. So you cannot use the registry to keep objects from being garbage-collected across snapshots. To do that, use the normal Smalltalk techniques (such as keeping the object in a class variable) and re-register your slots when the snapshot restarts.

An object that has a reference in the registry, however, will not be garbage-collected while the Object Engine is running — so it is important to store nil in a registry slot when you are done with it. Failure to do this can cause a space leak.

Restrictions

The Object Engine access functions cannot be called from 16-bit DLLs compiled for MS-Windows. Libraries must be compiled for 32-bit operation to use the Object Engine access interface.

Object Engine Access Overview

The following section lists the Object Engine access protocol. The first column indicates the desired operation, the second column indicates the appropriate function to use.

Table 27: Class conversion

Operation	Function
nil	oeNil()

Operation	Function
true	oeTrue()
false	oeFalse()
SmallInteger class	oeSmallIntegerClass()
Character class	oeCharacterClass()
Float class	oeFloatClass()
Double class	oeDoubleClass()
Point class	oePointClass()
Array class	oeArrayClass()
Semaphore class	oeSemaphoreClass()
ByteString class	oeByteStringClass()
LargePositiveIntegerClass	oeLargePositiveIntegerClass()
LargeNegativeIntegerClass	oeLargeNegativeIntegerClass()
ByteArray class	oeByteArrayClass()
TwoByteString class	oeTwoByteStringClass()
TwoByteSymbol class	oeTwoByteSymbolClass()

Table 28: C conversion

To convert the following:	Use the function:
C string to a Smalltalk string	oeCopyCtoOEstring()
C byte array to a Smalltalk byte object	oeCopyCtoOEbytes()
C integer array to Smalltalk Array	oeCopyCtoOEintArray()
C float array to a Smalltalk Array	oeCopyCtoOEfloatArray()
C integer to Smalltalk Integer	oeCtoOEint()
C double to a Smalltalk Double	oeCtoOEdouble()
C float to a Smalltalk Float	oeCtoOEfloat()
C boolean to a Smalltalk Boolean	oeCtoOEbool()
C character to a Smalltalk Character	oeCtoOEchar()

Table 29: Smalltalk conversion

To convert the following:	Use the function:
Smalltalk String to a C string	oeCopyOEtoCstring()
Smalltalk Byte Array to a C byte array	oeCopyOEtoCbytes()
Smalltalk Integer array to C integer array	oeCopyOEtoCintArray()

To convert the following:	Use the function:
Smalltalk Float array to C float array	oeCopyOEtoCfloatArray()
Smalltalk Integer to C integer	oeOEToCint()
Smalltalk Float to C float	oeOEToCfloat()
Smalltalk Double to C double	oeOEToCdouble()
Smalltalk Character to C character	oeOEToCchar()
Smalltalk Boolean to C boolean	oeOEToCbool()

Table 30: Data type functions

To test for this in Smalltalk:	Use this function:
Character	oeIsCharacter()
String	oeIsString()
Integer	oeIsInteger()
Float	oeIsFloat()
Double	oeIsDouble()
Array of Integers	oeIsArrayOfInteger()
Array of Floats	oeIsArrayOfFloat()
Byte Array	oeIsByteArray()
Byte-like	oeIsByteLike()
Boolean	oeIsBoolean()
Immediate	oeIsImmediate()
Class check	oeIsKindOf()
Valid Object	oeCouldBeOop()

Table 31: Allocation functions

To allocate this in Smalltalk:	Use this function:
String	oeAllocString()
Byte array	oeAllocByteArray()
Array	oeAllocArray()
Fixed-size object	oeAllocFsObject()
Variable-size object	oeAllocVsObject()
Fixed-space object	oeAllocVsObjectInFixedSpace()
New empty slot in the System Registry	oeAllocRegistrySlot()

Table 32: Access functions

To access this:	Use this function:
Indexed variable	oeBasicAt()
Instance variable	oeInstVarAt()
Indexed byte	oeByteAt()
Indexed float	oeFloatAt()
Indexed double	oeDoubleAt()
Registry Handle to object	oeRegisteredHandleAt()
Object’s instance variable size	oeInstVarSize()
Object’s indexed variable size	oeIndexVarSize()
Object’s data pointer	oeOopDataPtr()

Table 33: Storage functions

To store into this:	Use this function:
Indexed variable	oeBasicAtPut()
Instance variable	oeInstVarAtPut()
Indexed byte	oeByteAtPut()
Indexed float	oeFloatAtPut()
Registry Handle to object	oeRegisteredHandleAtPut()

Table 34: Send message functions

To send this:	Use this function:
Unary message	oeSendMessage0()
One-argument message	oeSendMessage1()
Two-argument message	oeSendMessage2()
Three-argument message	oeSendMessage3()
Multi-argument message	oeSendMessageMany()
Unary message with C string selector	oeCSendMessage0()
One-argument message with C string selector	oeCSendMessage1()
Two-argument message with C string selector	oeCSendMessage2()
Three-argument message with C string selector	oeCSendMessage3()
Multi-argument message with C string selector	oeCSendMessageMany()

Table 35: Computation functions

To perform long computations:	Use this function:
At the start of the long computation	oeStartLongComputation()
At the end of the long computation	oeFinishLongCompuation()

Table 36: Error condition function

To test this error condition:	Use this function:
To retrieve the last error code	oeGetErrorCode()
Function failure	oeFail()

Table 37: Initialization of statically linked code

To perform this action:	Use this function:
To perform initialization of statically linked code	oeInstall()
To define statically-linked entry points	oeInitLinkRegistry()

Table 38: Interrupt handling

To register an interrupt handler:	Use this function:
To set up an interrupt handler	oeInstallPollHandler()
To post an interrupt for the poll handler	oePostInterrupt()

Object Engine Access Reference

oeAllocArray

```
oeAllocArray(
OEoop
  oeInitValue,
 long
  numElements);
```

The `oeAllocArray` function allocates an instance of an `Array`.

Parameters

oeInitValue: Identifies the object used to initialize each element of the new array.

numElements: Identifies the number of object slots in the new string.

Return Value

Returns a new `Array` instance.

Comments

If the allocation fails, returns `OEnonOop` with the error code `OEerrorAllocationFailed`. If the argument **numElements** is invalid, the error code is `OEerrorCrange`. If the object **oeInitValue** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeBasicAt`, `oeBasicAtPut`, `oeInstVarAt`, `oeInstVarAtPut`

oeAllocByteArray

```
OEoop oeAllocByteArray(
OEbyte
  initByteValue,
long
  numElements);
```

The `oeAllocByteArray` function allocates an instance of a `ByteArray`.

Parameters

initByteValue: Identifies the byte that should be used to initialize each element of the new string.

numElements: Identifies the number of bytes in the new string.

Return Value

Returns a new `ByteArray` instance.

Comments

If the allocation fails, returns OEnonOop with the error code OEerrorAllocationFailed. If the argument **numElements** is negative, the error code is OEerrorCrange. If the object **initByteValue** is invalid, the error code is OEerrorNonOop.

See Also

oeIsByteArray, oeCopyOEtoCbytes, oeCopyCtoOEbytes, oeBasicAt, oeBasicAtPut, oeInstVarAt, oeInstVarAtPut, oeByteAt, oeByteAtPut

oeAllocFsObject

```
OEoop oeAllocFsObject(
OEoop oeClass);
```

The oeAllocFsObject function allocates an instance of a fixed-sized class. A fixed-sized class is any class that cannot have a variable number of instance variables.

Parameters

oeClass: Identifies the fixed-sized class to instantiate.

Return Value

Returns a new object instance.

Comments

If the allocation fails, returns OEnonOop with the error code OEerrorAllocationFailed. If the argument **oeClass** does not represent a fixed-sized class, the error code is OEerrorWrongClass. If the object **oeClass** is invalid, the error code is OEerrorNonOop.

See Also

oeInstVarAt, oeInstVarAtPut, oeBasicAt

oeAllocRegistrySlot

```
OEint oeAllocRegistrySlot(void);
```

The `oeAllocRegistrySlot` function allocates a new empty slot in the system registry.

Parameters

None

Return Value

Returns an identifier representing the allocated system registry slot.

Comments

Registry slots are a limited resource, and cannot be recycled (while the Object Engine is running), so use them sparingly. To reserve a registry slot, call:

```
static OEint slot;
slot = oeAllocRegistrySlot();
```

Any slots you allocate (and the references stored in them) are discarded in a snapshot file. Do not use the registry to keep objects from being garbage collected across snapshots. This can be done by using the normal Smalltalk programming techniques (such as keeping the object in a class variable) and re-register your slots when the snapshot starts up.

An object that has a reference in the registry, however, will not be garbage collected while the Object Engine is running so it is a good idea to store `nil` in a registry slot when you are finished with it.

If `oeAllocRegistrySlot` fails, it returns zero with the error code `OEallocationFailed`, which indicates a registry slot could not be allocated.

See Also

`oeRegisteredHandleAtPut`, `oeRegisteredHandleAt`

oeAllocString

```
OEoop oeAllocString(
 OEchar
  initCharValue,
 long
  numElements);
```

The `oeAllocString` function allocates an instance of a `ByteString`.

Parameters

initCharValue: Identifies the character that should be used to initialize each element of the new string.

numElements: Identifies the number of characters in the new string.

Return Value

Returns a new `ByteString` instance.

Comments

If the allocation fails, returns `OEnonOop` with the error code `OEerrorAllocationFailed`. If the argument **numElements** is negative, the error code is `OEerrorCrange`.

See Also

`oeIsString`, `oeCopyOEtoCstring`, `oeCopyCtoOEstring`, `oeBasicAt`, `oeBasicAtPut`, `oeInstVarAt`, `oeInstVarAtPut`, `oeByteAt`, `oeByteAtPut`

oeAllocVsObject

```
OEoop oeAllocVsObject(
 OEoop
  oeClass,
 OEint
  size);
```

The oeAllocVsObject function allocates an instance of a variable-sized class. A variable-sized class is any class that can have a variable number of variables capable of being indexed.

Parameters

oeClass: Identifies the class to instantiate.

size: Indicates the number of elements to allocate in the new object.

Return Value

Returns a new object instance.

Comments

If the allocation fails, returns OEnonOop with the error code OEerrorAllocationFailed. If the argument **oeClass** does not represent a variable-sized class, the error code is OEerrorWrongClass. If the argument **size** is negative, the error code is OEerrorCrange. If the object **oeClass** is invalid, the error code is OEerrorNonOop.

See Also

oeAllocVsObjectInFixedSpace, oeInstVarAt, oeInstVarAtPut, oeBasicAt, oeBasicAtPut

oeAllocVsObjectInFixedSpace

```
OEoop oeAllocVsObjectInFixedSpace(
 OEoop
  oeClass,
 OEint
  size);
```

The oeAllocVsObjectInFixedSpace function allocates an instance of a variable-sized class. A variable-sized class is any class that can have a variable number of variables capable of being indexed. This function allocates the object in Fixed Space. The class of the requested object must be compatible with such allocations.

Parameters

oeClass: Identifies the class to instantiate.

size: Indicates the number of elements to allocate in the new object.

Return Value

Returns a new object instance.

Comments

If the allocation fails, returns `OEnonOop` with the error code `OEerrorAllocationFailed`. If the argument **oeClass** does not represent a variable-sized class, the error code is `OEerrorWrongClass`. If the argument **size** is negative, the error code is `OEerrorCrange`. If `the object` **oeClass** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeAllocVsObject`, `oeInstVarAt`, `oeInstVarAtPut`, `oeBasicAt`, `oeBasicAtPut`

oeBasicAt

```
OEoop oeBasicAt(
 OEoop
  oeObject,
 OEint
  index);
```

The `oeBasicAt` function returns an arbitrary element within a variable-sized object.

Parameters

oeObject: Identifies the object to access.

index: Indicates the index of the object to retrieve.

Return Value

Returns the element in the `index` position of `oeObject`.

Comments

If `oeBasicAt` fails, returns `OEnonOop`. If the argument **oeObject** is not a variable-sized object, the error code is `OEerrorWrongClass`. If the argument **index** is invalid, the error code is `OEerrorCrange`. If **index** is larger than the size of **oeObject**, the error code is `OEerrorOOEerrorObjectTooSmall`. If the object **oeObject** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeAllocVsObject`, `oeInstVarAt`, `oeInstVarAtPut`, `oeBasicAtPut`

oeBasicAtPut

```
OEoop oeBasicAtPut(
 OEoop
  oeObject,
 OEint
  index
 OEoop
  oeOopToBePut);
```

The `oeBasicAtPut` function replaces an arbitrary element within a variable-sized object.

Parameters

oeObject: Identifies the object whose element in the **index** position is replaced.

oeOopToBePut: Identifies the object to placed into the **index** location in the position in **oeObject**.

index: Indicates the index of the object to replace.

Return Value

None

Comments

If oeBasicAtPut fails, returns OEnonOop with the error code OEerrorWrongClass if the argument **oeObject** is not a valid object. If the argument **index** is less than one, the error code is OEerrorCrange. If **index** is larger than the size of **oeObject**, the error code is OEerrorObjectTooSmall. If **oeObject** or **oeOopToBePut** are invalid handles, the error code is OEerrorNonOop.

See Also

oeAllocVsObject, oeInstVarAt, oeInstVarAtPut, oeBasicAtPut

oeByteAt

```
OEoop oeByteAt(
 OEoop
  oeByteLikeObject,
 OEint
  index,
 OEbyte
  *lpByte);
```

This function retrieves a byte from a byte-like object.

Parameters

oeByteLikeObject: Identifies the object whose byte in the **index** position is retrieved.

index: Indicates the index of the byte to retrieve.

lpByte: Pointer to the location where the fetched byte is to be placed.

Return Value

Returns the argument, oeByteLikeObject.

Comments

If oeByteAt fails, returns OEnonOop with the error code OEerrorWrongClass if the argument **oeByteLikeObject** is not a valid byte-like object. If the argument **index** is less than one, the error code is `OEerrorCrange`. If **index** is larger than the size of **oeByteLikeObject**, the error code is `OEerrorObjectTooSmall`. If the handle **oeByteLikeObject** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeAllocFsObject`, `oeInstVarAt`, `UPinstVarAtPut`, `oeBasicAt`, `oeBasicAtPut`

oeByteAtPut

```
OEoop oeByteAtPut(
 OEoop
  oeByteLikeObject,
 OEint
  index,
 OEbyte
  oeUpByte);
```

The `oeByteAtPut` function modifies a byte within a byte-like object.

Parameters

oeByteLikeObject: Identifies the object whose byte in the **index** position is replaced.

index: Indicates the index of the byte to retrieve.

oeUpByte: Indicates the byte to be placed at the **index** location of the object represented by **oeByteLikeObject**.

Return Value

None

Comments

If `oeByteAtPut` fails, returns `OEnonOop` with the error code `OEerrorWrongClass` if the argument **oeByteLikeObject** is not a valid byte-like object. If the argument **index** is less than one, the error code is `OEerrorCrange`. If **index** is larger than the size of **oeByteLikeObject**, the error code is `OEerrorObjectTooSmall`. If the object **oeByteLikeObject** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeAllocFsObject`, `oeInstVarAt`, `UPinstVarAtPut`, `oeBasicAt`, `oeBasicAtPut`

oeClass

```
OEoop oeClass(OEoop oeObject);
```

The `oeClass` function returns the class of an object.

Parameters

oeObject: Identifies the object whose class will be retrieved.

Return Value

Returns a handle representing the class of the argument **oeObject**.

Comments

If `oeClass` fails, returns `OEnonOop` with the error code `OEerrorNonOop` if the object **oeObject** is invalid.

See Also

`oeAllocVsObject`, `oeAllocFsObject`

oeClassType

```
OEclassID oeClassType(OEoop oeClass);
```

The `oeClassType` function returns an identifier indicating the class type of the argument.

Parameters

oeClass: Identifies the class object whose class type will be returned.

Return Value

Returns an identifier representing the class type of the class object represented by the handle **oeClass**.

Comments

The following indicates the possible return values:

```
typedef enum {
 OEnotAClass = 0,
 OEfixedSizeClass = 1,
 OEvariableSizeClass = 2
} OEclassID;
```

Returns `OEnotAClass` if the object **oeClass** is a valid, or if **oeClass** is a valid object but not a class.

See Also

`oeClass`, `oeFail`

oeCopyCtoOEbytes

```
OEoop oeCopyCtoOEbytes(
 OEoop
  oeByteObject,
 OEbyte
  *lpBytes,
 OEint
  aCount,
 OEint
  startingAt,
 OEint
  *lpActualCount);
```

The `oeCopyCtoOEbytes` function copies an array of bytes into a Smalltalk byte object.

Parameters

oeByteObject: Identifies the Smalltalk byte object that serves as the destination of the copy action.

lpBytes: Points to the array of bytes that is copied into `OEbyteObject`.

aCount: Indicates the number of bytes to copy.

startingAt: Indicates the starting index within `OEbyteObject` that the copy should take place. Note that the index is 1-based, not 0-based.

lpActualCount: Points to the location where the actual number of bytes copied is placed.

Return Value

Returns the argument, **oeByteObject**.

Comments

If `oeCopyCtoOEbytes` fails, returns `OEnonOop` with the error code `OEerrorWrongClass` if **oeByteObject** is not a valid object. If **startingAt** is out of range the error code is `OEerrorCrange`. If the object **oeByteObject** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeCopyOEtoCbytes`, `oeIsByteArray`

oeCopyCtoOEfloatArray

```
OEoop oeCopyCtoOEfloatArray(
 OEoop
  oeArray,
 OEfloat
  *lpFloats,
 OEint
  aCount,
 OEint
  startingAt,
 OEint
  *lpActualCount);
```

The `oeCopyCtoOEfloatArray` function copies an array of C floating-point numbers into an `Array` object.

Parameters

oeArray: Identifies the `Array` object that serves as the destination of the copy.

lpFloats: Points to the array of C floating-point numbers.

aCount: Indicates the number of floating-point numbers to copy.

startingAt: Indicates the starting index within **oeArray** that the copy should take place. Note that the index is 1-based, not 0-based.

lpActualCount: Points to the location where the actual number of bytes copied will be placed.

Return Value

Returns the argument, **oeArray**.

Comments

If `oeCopyCtoOEfloatArray` fails, it returns directly to the caller. The function fails with the error code `OEerrorWrongClass` if **oeArray** is not a valid Array object. If **startingAt** is out of range, the error code is `OEerrorCrange`. If the object **oeArray** is invalid `OEerrorNonOop`.

See Also

`oeCopyOEtoCfloatArray`, `oeIsArrayOfFloat`

oeCopyCtoOEintArray

```
OEoop oeCopyCtoOEintArray(
 OEoop
  oeArray,
 OEint
  *lpInts,
 OEint
  aCount,
 OEint
  startingAt,
 OEint
  *lpActualCount);
```

The oeCopyCtoOEintArray function copies an array of C integers into an Array object.

Parameters

oeArray: Identifies the Array object that serves as the destination of the copy action.

lpInts: Points to the array of C integers.

aCount: Indicates the number of integers to copy.

startingAt: Indicates the starting index within **oeArray** that the copy should take place. The index is 1-based, not 0-based.

lpActualCount: Points to the integer where the actual number of bytes copied should be placed.

Return Value

Returns the argument, **oeArray**.

Comments

If oeCopyCtoOEintArray fails, it returns OEnonOop with the error code OEerrorWrongClass if the object **oeArray** is not valid. If **startingAt** is out of range, the error code is OEerrorCrange. If **oeArray** is an invalid handle, the error code is OEerrorNonOop. If any of the integers pointed to by **lpInts** are out of range, the error code is OEerrorCrange.

Only integers in the range -229 to 229 - 1 are correctly converted.

See Also

oeCopyOEtoCintArray, oeIsArrayOfInteger

oeCopyCtoOEstring

```
OEoop oeCopyCtoOEstring(
 OEoop
  oeString,
 OEchar
  *lpszString,
 OEint
 aCount,
OEint
 startingAt,
OEint
 *lpActualCount);
```

The `oeCopyCtoOEstring` function copies a C string (pointer to a null terminated array of character elements) into a `ByteString`.

Parameters

oeString: Identifies the `String` object that serves as the destination of the copy action.

lpszString: Points to the null-terminated string that is copied into **oeString**.

aCount: Indicates the number of characters to copy.

startingAt: Indicates the starting index within **oeString** that the copy should take place. Note that the index is 1-based, not 0-based.

lpActualCount: Points to the integer where the actual number of bytes copied is placed.

Return Value

Returns the argument, **oeString**.

Comments

If `oeCopyCtoOEstring` fails, returns `OEnonOop` with the error code `OEerrorWrongClass` if **oeString** is not a valid `ByteString` object. If **startingAt** is out of range, the error code is `OEerrorCrange`. If the object **oeString** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeCopyOEtoCstring`, `oeIsString`

oeCopyOEtoCbytes

```
OEoop oeCopyOEtoCbytes(
 OEoop
 oeByteObject,
OEbyte
 *lpBytes,
OEint
 aCount,
OEint
 startingAt,
OEint
 *lpActualCount);
```

The oeCopyOEtoCbytes function copies a variable length byte object into a C byte array.

Parameters

oeByteObject: Identifies the byte object that serves as the source of the copy.

lpBytes: Points to the memory location where the bytes are placed.

aCount: Indicates the number of bytes to copy.

startingAt: Indicates the starting index within **oeByteObject** that the copy should take place. Note that the index is 1-based, not 0-based.

lpActualCount: Points to the actual number of bytes copied.

Return Value

Returns the argument **oeByteObject**.

Comments

If oeCopyOEtoCbytes fails, returns OEnonOop with the error code OEerrorWrongClass if **oeByteObject** is not a valid byte-like object. If **startingAt** is out of range, the error code is OEerrorCrange. If the object **oeByteObject** is invalid, the error code is OEerrorNonOop.

See Also

oeCopyCtoOEbytes, oeIsByteArray

oeCopyOEtoCfloatArray

```
OEoop oeCopyOEtoCfloatArray(
 OEoop
  oeFloatArray,
 OEfloat
  *lpFloats,
 OEint
  aCount,
 OEint
  startingAt,
 OEint
  *lpActualCount);
```

The `oeCopyOEtoCfloatArray` function copies a Smalltalk array of floating-point numbers into a C array of floating-point numbers.

Parameters

oeFloatArray: Identifies the Smalltalk float array object that serves as the source of the copy action.

lpFloats: Points to the memory location where the floating point numbers are placed.

aCount: Indicates the number of floats to copy.

startingAt: Indicates the starting index within **oeFloatArray** that the copy should take place. Note that the index is 1-based, not 0-based.

lpActualCount: Points to the number indicating the actual number of bytes copied.

Return Value

Returns the argument **oeFloatArray**.

Comments

If `oeCopyOEtoCfloatArray` fails, returns `OEnonOop` with the error code `OEerrorWrongClass` if **oeFloatArray** is not a valid Array object. If **startingAt** is out of range, the error code is `OEerrorCrange`. If the object **oeFloatArray** is invalid, the error code is `OEerrorNonOop`. If any of the elements in **oeFloatArray** are not Smalltalk Float objects, the error code is OEerrorWrongClass.

See Also

oeCopyCtoOEfloatArray, oeIsFloatArray

oeCopyOEtoCintArray

```
OEoop oeCopyOEtoCintArray(
 OEoop
  oeIntArray,
 OEint
  *lpInts,
 OEint
  aCount,
 OEint
  startingAt,
 OEint
  *lpActualCount);
```

The oeCopyOEtoCintArray function copies an array of SmallIntegers into a C integer array.

Parameters

oeintArray: Identifies the SmallInteger array object that serves as the source of the copy action.

lpInts: Points to the memory location where the integers are placed.

aCount: Indicates the number of integers to copy.

startingAt: Indicates the starting index within **oeIntArray** that the copy should take place. Note that the index is 1-based, not 0-based.

lpActualCount: Points to the actual number of bytes copied.

Return Value

Returns the argument **oeIntArray**.

Comments

If `oeCopyOEtoCintArray` fails, returns `OEnonOop` with the error code `OEerrorWrongClass` if **oeIntArray** is not a valid Array object. If **startingAt** is out of range, the error code is `OEerrorCrange`. If the object **oeIntArray** is invalid, the error code is `OEerrorNonOop`. If any of the elements in **oeIntArray** are not `SmallInteger` objects, the error code is `OEerrorWrongClass`.

See Also

`oeCopyCtoOEintArray`, `oeIsIntegerArray`

oeCopyOEtoCstring

```
OEoop oeCopyOEtoCstring(
 OEoop
  oeString,
 OEchar
  *lpszString,
 OEint
  aCount,
 OEint
  startingAt,
 OEint
  *lpAcutalCount);
```

The `oeCopyOEtoCstring` function copies a `ByteString` object into a C string.

Parameters

oeString: Identifies the `ByteString` object that serves as the source of the copy action.

lpszString: Points to the memory location where the string is placed.

aCount: Indicates the number of characters to copy.

startingAt: Indicates the starting index within **oeString** that the copy should take place. Note that the index is 1-based, not 0-based.

lpActualCount: Points to the integer indicating the actual number of bytes copied.

Return Value

Returns the argument **oeString**.

Comments

If `oeCopyOEtoCstring` fails for any reason, returns `OEnonOop` with the error code `OEerrorWrongClass` if **oeString** is not a valid `ByteString` object. If **startingAt** is out of range, the error code is `OEerrorCrange.` If the object **oeString** is invalid `OEerrorNonOop`.

Note that the resulting C string is not null-terminated.

See Also

`oeCopyCtoOEstring`, `oeIsString`

oeCSendMessage

```
OEoop oeCSendMessage0(
 OEoop
  oeReceiver,
 char
  *szSelector,
 OEoop
  *poeKeep,
 OEoop
  oeFailure);OEoop oeCSendMessage1(
 OEoop
  oeReceiver,
 char
  *szSelector,
 OEoop
  oeArg1,
 OEoop
  *poeKeep,
 OEoop
  oeFailure);OEoop oeCSendMessage2(
 OEoop
  oeReceiver,
 char
  *szSelector,
 OEoop
  oeArg1,
OEoop
 oeArg2,
OEoop
 *poeKeep,
OEoop
 oeFailure);OEoop oeCSendMessage3(
OEoop
 oeReceiver,
char
 *szSelector,
OEoop
 oeArg1,
OEoop
 oeArg2,
OEoop
 oeArg3,
OEoop
 *poeKeep,
OEoop
 oeFailure);OEoop oeCSendMessageMany(
OEoop
 oeReceiver,
char
 *szSelector,
OEoop
 oeArgArray,
OEoop
 *poeKeep,
OEoop
 oeFailure);
```

The `oeCSendMessage` function set is used to send unary, binary and keyword messages to Smalltalk objects.

Parameters

oeReceiver: Receiver of the message.

szSelector: Pointer to a null-terminated array of characters representing the message selector.

oeArg1: First argument for binary and keyword messages.

oeArg2: Second argument for keyword messages.

oeArg3: Third argument for keyword messages.

oeArgArray: Array of message arguments for keyword messages.

poeKeep: Pointer to a handle representing a Smalltalk object that should stay valid after the message send completes.

oeFailure: Object handle representing the value that is returned if any error occurs during the message send.

Return Value

After evaluating the Smalltalk message send, the function returns that value.

Comments

The functions listed in this section are identical to the functions described in `OESendMessage`, except that the message selector is represented by a null-terminated sequence of characters rather than a `Symbol` object handle. This function set permits your C code to manufacture the selector name instead of being restricted to using a selector passed down from Smalltalk. The differences between this function set and the `OESendMessage` function set are:

- The method name has the letter C in its prefix, as in `oeCSendMessage0()`
- The **oeSelector** argument type is `char *` instead of `OEoop`

Each function returns the value returned by the message-send. If the message-send fails, or if an argument is invalid, the function returns immediately to the calling C function.

See Also

`oeSendMessage`

oeCtoOEbool

```
OEoop oeCtoOEbool(OEbool oeBool);
```

The `oeCtoOEbool` function converts a C boolean into a `Boolean` object (true or false.)

Parameters

oeBool: Identifies the C boolean to be converted.

Return Value

Returns the `Boolean` object (true or false) if the conversion succeeds, otherwise the return value is `OEnonOop`.

See Also

`oeCopyOEtoCbool`, `oeIsBoolean`

oeCtoOEchar

```
OEoop oeCtoOEchar(OEchar oeChar);
```

The `oeCtoOEchar` function converts a C character into a `Character` object.

Parameters

oeChar: Identifies the C character value to be converted.

Return Value

Returns the `Character` object if the conversion succeeds, otherwise the return value is `OEnonOop`.

Comments

Multi-byte characters are currently not supported.

See Also

`oeCopyOEtoCchar`, `oeIsCharacter`, `oeIsImmediate`

oeCtoOEdouble

```
OEoop oeCtoOEdouble(OEdouble oeDouble);
```

The `oeCopyCtoOEdouble` function converts a C double precision floating point number into a `Double` object.

Parameters

oeDouble: Identifies the C double precision floating point number to be converted.

Return Value

Returns the Double object.

Comments

If oeCopyCtoOEdouble fails, returns OEnonOop with the error code OEerrorAllocationFailed if the double object could not be allocated.

See Also

oeCopyOEtoCdouble, oeIsDouble

oeCtoOEfloat

```
OEoop oeCtoOEfloat(upFlaot oeFloat);
```

The oeCopyCtoOEfloat function converts a C floating point number into a Float object.

Parameters

oeFloat: Identifies the C floating-point number to be converted.

Return Value

Returns the Float object.

Comments

If oeCopyCtoOEfloat fails, returns OEnonOop with the error code OEerrorAllocationFailed if the float object could not be allocated.

See Also

oeCopyOEtoCfloat, oeIsFloat

oeCtoOEint

```
OEoop oeCtoOEint(OEint oeInt);
```

The `oeCopyCtoOEint` function converts a C integer into a `SmallInteger`.

Parameters

oeInt: Identifies the C integer value to be converted.

Return Value

Returns the `SmallInteger` object.

Comments

The function `oeCopyCtoOEint` fails with the error code `OEerrorCrange` if the argument **oeInt** is too large or small.

This function successfully converts integers in the range -229 to 229 - 1. This is because of the way Smalltalk represents `SmallIntegers`.

See Also

`oeCopyOEtoCint`, `oeIsInteger`

oeDoubleAt

```
OEoop oeDoubleAt(
 OEoop
   oeArray,
 OEint
   index,
 OEdouble
   *lpDouble);
```

The `oeDoubleAt` function retrieves a `Double` object from an `Array`.

Parameters

oeArray: Identifies the array whose double element in the **index** position is retrieved.

index: Indicates the index of the double in the argument **oeArray**.

lpDouble: Points to the location where the double object is placed.

Return Value

Returns the argument **oeArray**.

Comments

The double object retrieved from the array is converted into a C double before the function returns.

If `oeDoubleAt` fails, it returns `OEnonOop` with the error code `OEerrorWrongClass` if the argument **oeArray** is not an array. If the object **oeArray** is not valid, the error code is `OErrorNonOop`. If the object at the **index** location is not a double, the error code is `OEerrorWrongClass`. If the argument **index** is less than one, the error code is `OEerrorCrange`. If **index** is larger than the size of **oeArray**, the error code is `OEerrorObjectTooSmall`.

See Also

`oeIsDouble`, `oeOEToCdouble`, `oeCtoOEdouble`

oeFail

```
void oeFail(OEint failCode);
```

The `oeFail` function forces a failure return directly to Smalltalk.

Parameters

failCode: Identifies the value that should be returned.

Return Value

None. The function does not return to the C caller; instead, control is returned directly to Smalltalk.

Comments

If a C function call from a Smalltalk method fails, and if the invoking method contains Smalltalk code after the `<C: ...>` statement, then the Smalltalk failure code will be evaluated. Common causes for failure may be the result of the type checking and type conversion routines.

For a complete listing of the error codes that will be posted by this routine, see Failure Codes.

Once control is passed back to Smalltalk, an error is available in `_errorCode`. This hidden temporary variable in the calling method will contain a `SystemError`. The `parameter` field of this `SystemError` object is set to the argument passed to `oeFail()`, if this function is called directly.

See Also

`oeGetErrorCode`

oeFloatAt

```
OEoop oeFloatAt(
 OEoop
  oeArray,
 OEint
  index,
 OEfloat
  *lpFloat);
```

The `oeFloatAt` function retrieves a `Float` object from an `Array`.

Parameters

oeArray: Identifies the array whose float element in the **index** position is retrieved.

index: Indicates the index of the float in the argument **oeArray**.

lpFloat: Points to the location where the float is to be placed.

Return Value

Returns the argument **oeArray**.

Comments

The float object retrieved from the array is converted into a C float before the function returns.

If `oeFloatAt` fails, returns `OEnonOop` with the error code `OEerrorWrongClass` if the argument **oeArray** is not an array. If the object **oeArray** is not valid, the error code is `OEerrorNonOop`. If the object at the **index** location is not a float, the error code is `OEerrorWrongClass`. If the argument **index** is less than one, the error code is `OEerrorCrange`. If **index** is larger than the size of **oeArray**, the error code is `OEerrorObjectTooSmall`.

See Also

`oeIsArrayOfFloat`, `oeCopyOEtoCfloatArray`, `oeCopyCtoOEfloatArray`, `oeFloatAtPut`

oeFloatAtPut

```
OEoop oeFloatAtPut(
 OEoop
  oeArray,
 OEint
  index,
 OEfloat
  aFloat);
```

The `oeFloatAtPut` function places a C float in an `Array`.

Parameters

oeArray: Identifies the array whose float element in the **index** position is modified.

index: Indicates the index in the argument **oeArray** to place the float.

aFloat: Indicates the float object to place into the **index** element of **oeArray**.

Return Value

Returns the argument **oeArray**.

Comments

The C float is first converted into a float object before being placed into the array.

If `oeFloatAtPut` fails for any reason, this function returns `OEnonOop`. If the float object could not be created when converting the C float object, the error code is `OEallocationFailed`. If the argument **oeArray** is not an array, the error code is `OEerrorWrongClass`. If the object **oeArray** is invalid, the error code is `OEerrorNonOop`. If the argument **index** is less than one, the error code is `OEerrorCrange`. If **index** is larger than the size of **oeArray**, the error code is `OEerrorObjectTooSmall`.

See Also

`oeIsArrayOfFloat`, `oeCopyOEtoCfloatArray`, `oeCopyCtoOEfloatArray`, `oeFloatAt`

oeGetErrorCode

```
OEerrorCodeID oeGetErrorCode(void);
```

The `oeGetErrorCode` function returns the `OEerrorCodeID` after an Object Engine function call failure.

Parameters

None

Return Value

Returns the enumerated value `OEerrorCodeID`; for details, you may consult the file `oeAPI.h` (this file is located in the `src` subdirectory of the DLL and C Connect release).

Comments

An Object Engine function call can fail for a number of different reasons, but typically there is a problem with one of the arguments, or else an allocation attempt failed.

See Also

`oeFail`

oeIndexVarSize

```
OEoop oeIndexVarSize(
 OEoop
  oeObject,
 OEint
  *lpIndexVarSize);
```

The `oeIndexVarSize` function returns the size of the variable portion of an object.

Parameters

oeObject: Identifies the object whose variable portion size is retrieved.

lpIndexVarSize: Points to the integer where the object size is placed.

Return Value

Returns the size of the variable portion of **oeObject** in the number of `OEoops` if **oeObject** represents a pointer object, or the number of bytes if **oeObject** is a byte-like object.

Comments

`oeIndexVarSize` places the variable portion size of **oeObject** into **lpIndexVarSize**. It is represented as the number of `OEoops` if **oeObject** is a pointer object, or the number of bytes if **oeObject** is a byte-like object.

If `oeIndexVarSize` fails, returns `OEnonOop`. If **anObject** represents an illegal or immediate object, the error code is `OEerrorNonOop`.

See Also

`oeAllocVsObject`

oeInitLinkRegistry

```
void oeInitLinkRegistry(void);
```

The `oeInitLinkRegistry` function is used to define statically linked entry points in a custom Object Engine.

Parameters

None

Return Value

None

Comments

DLL and C Connect can access functions and data either dynamically or statically linked to the Object Engine. When statically linked entry points are used, you must define a mapping between the entry point name and it actual address. Perform this with the function `oeRegisterSymbolAndHandle`.

This function is the location where you should place your `oeRegisterSymbolAndHandle` function calls.

Note: This function must be defined in your statically linked Object Engine.

See the section Static Linking for a description on building statically linked Object Engines.

See Also

`oeRegisterSymbolAndHandle`

oeInstall

```
char *oeInstall(void);
```

The `oeInstall` function is called to initialize a custom Object Engine.

Parameters

None

Return Value

Returns a pointer to a null-terminated string that is used by the Object Engine kernel when displaying the herald is returned.

Comments

`oeInstall` is called by the kernel of your custom Object Engine prior to loading the image and running Smalltalk code. Place any customized initialization code in this function.

Note: This function must be defined in your statically linked Object Engine.

See Also

`oeInitLinkRegistry`, `oeInstallPollHandler`

oeInstallPollHandler

```
OEvoidFuncPtr oeInstallPollHandler(OEvoidFuncPtr pollHandler);
```

The `oeInstallPollHandler` function registers a handler that is called when the function `oePostInterrupt` is invoked.

Parameters

pollHandler: Pointer to a function that is called when the function `oePostInterrupt` is invoked.

Return Value

Returns a pointer to the previously installed poll handler. If there was no previously installed poll handler, returns NULL.

Comments

In some situations, a Smalltalk application needs to respond to a condition that can only be detected asynchronously (such as a UNIX signal-handler, or a PC interrupt-handler). In such a case, it is not possible to directly access object memory (e.g., to post a signal condition), because the event might occur while the Object Engine is performing an operation that involves relocating objects. You could create a Smalltalk process that periodically calls a function to see whether that event has occurred (by examining a C static variable, perhaps, or by making an operating system call), but this constant polling can be inefficient or inconvenient.

As an alternative, call the support routine `oePostInterrupt()` from your asynchronous signal/interrupt-handler. This routine arranges to have a poll-handler called soon thereafter before the next backward branch (in any loop) or frame-building send (all sends build frames except those that simply return a variable or call a primitive). This is the only Object Engine support routine that you can call asynchronously.

The poll-handler must be registered previous to the interrupt, by inserting a call to `oeInstallPollHandler` in your `oeInstall` or `oeRegisterSymbolAndHandle` routine.

Your handler is only called once, no matter how many times `oePostInterrupt` is invoked in the interim. Posting an interrupt simply sets a flag, which is cleared only when your poll-handler is called.

The handler usually determines the event that occurred (again, by examining a C static variable or by making an OS call) and possibly signals a semaphore that was stored previously in the registry. The only Object Engine support routines that a handler can safely call are:

- `oeSignalSemaphore()`
- `oeRegisteredHandleAt()`
- `oeRegisteredHandleAtPut()`

On UNIX platforms, the Object Engine requires unimpeded access to certain signals. Specifically, your C functions should not establish signal handlers for the following signals:

- SIGIO or SIGPOLL
- SIGALRM
- SIGVTALRM
- SIGCHLD

In addition, functions should not make system calls that generate the last three signals listed above. Operations that generate SIGIO or SIGPOLL do not cause problems.

See Also

oePostInterrupt

oeInstVarAt

```
OEoop oeInstVarAt(
 OEoop
  oeObject,
 OEint
  index);
```

The oeInstVarAt function returns the value of an object's instance variable.

Parameters

oeObject: Identifies the object whose instance variable in the **index** position is retrieved.

index: Indicates the index of the instance variable to retrieve.

Return Value

Returns the object located at the **index** instance variable location of object indicated by **oeObject**. If an error occurs, the return value is OEnonOop.

Comments

If oeInstVarAt fails, returns OEnonOop. If the object **oeObject** is invalid, the error code is OEerrorWrongClass. If the argument **index** less than one, the error code is OEerrorCrange. If **index** is larger than the size of **oeObject**, the error code is OEerrorObjectTooSmall.

See Also

oeAllocFsObject, oeInstVarAtPut, oeBasicAt, oeBasicAtPut

oeInstVarAtPut

```
OEoop oeInstVarAtPut(
 OEoop
  oeObject,
 OEint
  index,
 OEoop
  oopToBePut);
```

The oeInstVarAtPut function replaces an object's instance variable with another object.

Parameters

oeObject: Identifies the object whose instance variable at the **index** position is replaced.

index: Indicates the index of the instance variable to replace.

oopToBePut: Identifies the object that replaces the instance variable at the **index** position of **oeObject**.

Return Value

Returns the argument **oeObject**.

Comments

If oeInstVarAtPut fails, returns OEnonOop. If the object **oeObject** is invalid, the error code is OEerrorWrongClass. If the argument **index** is less than 1, the error code is OEerrorCrange. If **index** is larger than the size of **oeObject**, the error code is OEerrorObjectTooSmall.

See Also

oeAllocFsObject, oeInstVarAt, oeBasicAt, oeBasicAtPut

oeInstVarSize

```
OEoop oeInstVarSize(
 OEoop
  oeObject,
 OEint
  *lpInstVarSize);
```

The oeInstVarSize function returns the number of named instance variables in an object.

Parameters

oeObject: Identifies the object whose named instances variable count is retrieved.

lpInstVarSize: Points to the location where the instance variable size is placed.

Return Value

Returns the argument **oeObject**.

Comments

If oeInstVarSize fails, returns OEnonOop with the error code OEerrorNonOop if the object **oeObject** is invalid.

See Also

oeAllocFsObject

oeIsArrayOfFloat

```
OEbool oeIsArrayOfFloat(OEoop oeObject);
```

The oeIsArrayOfFloat function tests if an object is an Array containing only Float objects.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid `Array` of `Floats`, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the object **oeObject** is invalid.

See Also

`oeCopyCtoOEfloatArray`, `oeCopyOEtoCfloatArray`

oeIsArrayOfInteger

```
OEbool oeIsArrayOfInteger(OEoop oeObject);
```

The `oeIsArrayOfInteger` function tests if an object is an `Array` containing only `SmallInteger` objects.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid `Array` of `SmallIntegers`.

Comments

If `oeIsArrayOfInteger` fails, it returns `OEFALSE`. If the object **oeObject** is invalid, the error code is `OEerrorNonOop`.

See Also

`oeCopyCtoOEintArray`, `oeCopyOEtoCintArray`

oeIsBoolean

```
OEbool oeIsBoolean(OEoop oeObject);
```

The oeIsBoolean function tests if an object is a Boolean object (true or false).

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

The return value is OETRUE if the argument is a valid Boolean object, otherwise it is OEFALSE.

Comments

Returns OEFALSE if the object **oeObject** is invalid.

See Also

oeCtoOEbool, oeOEToCbool

oeIsByteArray

```
OEbool oeIsByteArray(OEoop oeObject);
```

The oeIsByteArray function tests if an object is a ByteArray.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns OETRUE if the argument is a valid ByteArray object, otherwise it is OEFALSE.

Comments

Returns OEFALSE if the object **oeObject** is invalid.

See Also

`oeCopyCtoOEbytes`, `oeCopyOEtoCbytes`

oeIsByteLike

```
OEbool oeIsByteLike(OEoop oeObject);
```

The `oeIsByteLike` function tests if an object is byte-like. An object is considered byte-like if it is not an immediate object, and it does not contain object pointers.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid byte-like object, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the object `oeObject` is invalid.

See Also

`oeCopyCtoOEbytes`, `oeCopyOEtoCbytes`

oeIsCharacter

```
OEbool oeIsCharacter(OEoop oeObject);
```

The `oeIsCharacter` function tests if an object is a `Character`.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid `Character` object, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the object **oeObject** is invalid.

See Also

`oeCtoOEchar`, `oeOEToCchar`

oeIsDouble

```
OEbool oeIsDouble(OEoop oeObject);
```

The `oeIsDouble` function tests if an object is a `Double`.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid `Double` object, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the object **oeObject** is invalid.

See Also

`oeCtoOEdouble`, `oeOEToCdouble`

oeIsFloat

```
OEbool oeIsFloat(OEoop oeObject);
```

The `oeIsFloat` function tests if the argument is a `Float` object.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid `Float` object, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the object **oeObject** is invalid.

See Also

`oeCtoOEfloat`, `oeOEToCfloat`

oeIsImmediate

```
OEbool oeIsImmediate(OEoop oeObject);
```

The `oeIsImmediate` function tests if an object is an immediate Smalltalk object.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid immediate Smalltalk object, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the object **oeObject** is invalid.

Immediate objects are objects that can be represented directly in the object pointer, and therefore do not require a separate object header and object data. Immediate objects save memory and processing time. VisualWorks implements character and `SmallInteger` objects as immediate objects.

See Also

oeIsCharacter, oeIsInteger

oeIsInteger

```
OEbool oeIsInteger(OEoop oeObject);
```

The oeIsInteger function tests if an object is a SmallInteger.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns OETRUE if the argument is a valid SmallInteger object, otherwise it is OEFALSE.

Comment

Returns OEFALSE if the object **oeObject** is invalid object.

See Also

oeCtoOEint, oeOEToCint

oeIsKindOf

```
OEbool oeIsKindOf(
 OEoop
  oeObject,
 OEoop
  oeClass);
```

The oeIsKindOf function tests if an object is an instance of a particular Smalltalk class.

Parameters

oeObject: Identifies the Smalltalk object to test.

oeClass: Identifies the Smalltalk class object to test against **oeObject**.

Return Value

Returns `OETRUE` if **oeObject** is an instance of the Smalltalk class **oeClass**, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the objects **oeObject** or **oeClass** are invalid.

`oeIsKindOf` checks the class of **oeObject** to determine if it matches **oeClass** or any of **oeClass**'s superclasses.

See Also

`oeClass`

oeIsString

```
OEbool oeIsString(OEoop oeObject);
```

The `oeIsString` function tests if an object is a `ByteString`.

Parameters

oeObject: Identifies the Smalltalk object to test.

Return Value

Returns `OETRUE` if the argument is a valid `ByteString` object, otherwise it is `OEFALSE`.

Comments

Returns `OEFALSE` if the object **oeObject** is an invalid handle.

See Also

`oeCtoOEstring`, `oeToCstring`

oeNil

```
OEoop oeNil(void);
```

The `oeNil` returns a handle to the nil object.

Parameters

None

Return Value

Returns a handle to the nil object

See Also

`oeReturnNil`, `oeIsImmediate`

oePostInterrupt

```
void oePostInterrupt(void);
```

The oePostInterrupt function is called to arrange that if the user's poll handler is installed, that it will be called when the Object Engine next polls for events.

Parameters

None

Return Value

None

Comments

This routine is typically used by code that associates a host OS signalling mechanism with Smalltalk semaphores. When you need to set the flag for signalling a semaphore, this routine can be used to arrange to have the Object Engine poll-handler called. More specifically, the Object Engine will call the handler before the next backward branch (in any loop) or a frame-building send (all sends build frames except those that simply return a variable or call a primitive). This is the only Object Engine support routine that you can call asynchronously.

The poll-handler must be registered previous to the interrupt, by inserting a call to `oeInstallPollHandler` in your `oeInstall` or `oeRegisterSymbolAndHandle` routine. For additional details, see the discussion of `oeInstallPollHandler`.

See Also

`oeInstallPollHandler`

oeRegisteredHandleAt

```
OEoop oeRegisteredHandleAt(OEint registrySlot);
```

The `oeRegisteredHandleAt` function returns an object handle previously stored in the System Registry.

Parameters

registrySlot: Identifies the slot in the System Registry (previously allocated with a call to `oeAllocRegistrySlot`) whose value should be returned.

Return Value

Returns the object contained in the System Registry at the location specified by the argument **registrySlot**. If the argument is an invalid slot, the return value is `OEnonOop`.

Comments

The Object Engine maintains a System Registry of objects that must be used when making direct references to the object memory over the span of several function calls. This is needed because the Object Engine relocates objects during memory management operations, so direct references to object memory (such as `OEoops`) cannot normally persist across function calls.

To refer to objects over time, the Object Engine provides a facility to register indirect references to objects. These indirect references are indices in a table that the Object Engine memory manager keeps current.

Another important reason to use the registry is for referencing an object that was given to a function in a prior call. For example, imagine you have a function that was passed a Semaphore so it could be signalled later (by using `OEsignalSemaphore`). Your function can ask the registry for a permanent slot (it returns a table index if there is enough room). The function stores the Semaphore into the registry at that slot. You record the slot index in a static variable in the C code. Later, to reference the object, read the registry at the slot you recorded.

If **registrySlot** is an invalid System Registry slot number, `oeRegisteredHandleAt` fails, and returns `OEnonOop` with the error code `OEerrorCrange`.

See Also

`oeAllocRegistrySlot`, `oeRegisteredHandleAtPut`

oeRegisteredHandleAtPut

```
OEoop oeRegisteredHandleAtPut(
 OEint
  registrySlot,
 OEoop
  oeOopToPut);
```

The `oeRegisteredHandleAtPut` function places an object handle into a pre-allocated slot in the System Registry.

Parameters

registrySlot: Identifies the slot in the System Registry (previously allocated with a call to `OEallocRegistrySlot`) whose value should be modified.

oeOopToPut: Identifies the object handle to place into the System Registry at the slot indicated by **registrySlot**.

Return Value

Returns the argument `oeOopToPut` if the store succeeds.

Comments

Use this function to store an object handle into the System Registry so that it may be retrieved later using `oeRegisteredHandleAt`. This facility was created to store object handles that are needed between function invocations.

If `oeRegisteredHandleAtPut` fails, returns `OEnonOop` with the error code `OEerrorCrange` if **registrySlot** is an invalid System Registry slot number.

See Also

`oeAllocRegistrySlot`, `oeRegisteredHandleAt`

oeRegisterSymbolAndHandle

```
void oeRegisterSymbolAndHandle(
 char
  *lpszSymbol,
 void
  *symbolHandle);
```

The `oeRegisterSymbolAndHandle` function associates a string with a procedure or data entry point.

Parameters

lpszSymbol: Points to a null-terminated sequence of characters used during handle look-up.

symbolHandle: Points to the object (function or data variable) associated with the symbol **lpszSymbol**.

Return Value

None

Comments

Use this function to register statically linked Object Engine entry points (either a function or data variable) that you access using DLL and C Connect. Functions can be called and data can be accessed that is statically linked into Smalltalk's Object Engine. The approach to mapping a symbolic name to a function entry-point or data variable address is different for dynamically and statically linked entities, and hence the need to manually register the entry points you have linked to the Object Engine.

Consult the section Static Linking.

See Also

oeInitLinkRegistry

oeSendMessage

```
OEoop oeSendMessage0(
 OEoop
  oeReceiver,
 OEoop
  oeSelector,
 OEoop
  *poeKeep,
 OEoop
  oeFailure);OEoop oeSendMessage1(
 OEoop
  oeReceiver,
 OEoop
  oeSelector,
 OEoop
  oeArg1,
 OEoop
  oeKeep,
 OEoop
  oeFailure);OEoop oeSendMessage2(
 OEoop
  oeReceiver,
 OEoop
  oeSelector,
 OEoop
  oeArg1,
OEoop
 oeArg2,
OEoop
 *poeKeep,
OEoop
 oeFailure);OEoop oeSendMessage3(
OEoop
 oeReceiver,
OEoop
 oeSelector,
OEoop
 oeArg1,
OEoop
 oeArg2,
OEoop
 oeArg3,
OEoop
 *poeKeep,
OEoop
 oeFailure);OEoop oeSendMessageMany(
OEoop
 oeReceiver,
OEoop
 oeSelector,
OEoop
 oeArgArray,
OEoop
 *poeKeep,
OEoop
 oeFailure);
```

The `oeSendMessage` function set is used to send unary, binary and keyword messages to Smalltalk objects.

Parameters

oeReceiver: Receiver of the message.

oeSelector: Message selector.

oeArg1: First argument for binary and keyword messages.

oeArg2: Second argument for keyword messages.

oeArg3: Third argument for keyword messages.

oeArgArray: Argument list for keyword messages.

poeKeep: Pointer to a handle representing an object that should maintain valid after the message send completes.

oeFailure: Object handle representing the value that `oeSendMessage0` should return if any error occurs during the message send.

Return Value

Returns the value of evaluating the Smalltalk message send.

Comments

The Object Engine defines a set of five special message-passing functions. The first function is for unary message, and the other four are for one-, two-, three-, and multi-keyword messages. Each function takes arguments that identify the receiver of the message, the message selector, and the arguments (if any). These objects are Smalltalk objects.

Each function takes an argument, **poeKeep**, that is used to store pointers to objects that your function uses after the callback; it keeps them safe during any memory management operations that might occur during the callback. Note, however, that this strategy is only effective for pointers that are held by the original function; if a calling function anywhere in the C stack holds a pointer into Smalltalk memory, that pointer can be invalidated during a callback. For this reason, only use callbacks in functions that are invoked directly from Smalltalk, where the **oeKeep** mechanism can be used, rather than in a secondary function (such as one nested further down the C call chain).

Finally, an argument named **oeFailure** is used to store the object that is to be returned if the message-send fails.

Each function returns the value returned by the message-send. If the message-send fails, or if any argument is invalid, the function returns the **oeFailure** argument.

The **oeSelector** argument must be an instance of `Symbol`. For example, a valid selector argument for `oeSendMessage1` would be `#add:`. A parallel set of functions (see `oeCSendMessage`) enables you to specify the selector name as a null-terminated sequence of characters in effect, your C code can manufacture the selector name on the spot instead of being restricted to the selector that is passed down from Smalltalk.

See Also

`oeCSendMessage`

oeSignalSemaphore

```
upBool oeSignalSemaphore(OEoop oeSemaphore);
```

The `oeSignalSemaphore` is used to signal a `Semaphore` object.

Parameters

oeSemaphore: Identifies the `Semaphore` object to be signalled.

Return Value

Returns `OETRUE` if the `Semaphore` was successfully signalled. If **oeSemaphore** is an invalid `Semaphore`, or if the semaphore could not be signalled, returns `OEFALSE`.

See Also

`oeRegisteredHandleAt`, `oeRegisteredHandleAtPut`

oeOEToCbool

```
OEoop oeOEToCbool(
 OEoop
  oeBoolean,
 OEbool *lpBool);
```

The `oeOEToCbool` function converts a `Boolean` (true or false) object into a C boolean.

Parameters

oeBoolean: Identifies the `Boolean` object to convert.

lpBool: Points to the location where the C boolean is placed.

Return Value

Returns the argument **oeBoolean**.

Comments

If oeOEToCbool fails, returns OEnonOop. If **oeBoolean** does not represent a valid handle, the error code is OEerrorNonOop. If **oeBoolean** is not a boolean, the error code is OEerrorWrongClass.

See Also

oeCopyCtoOEbool, oeIsBoolean

oeOEToCchar

```
OEoop oeOEToCchar(
 OEoop
  oeCharacter,
 OEchar
  *lpChar);
```

The oeOEToCchar function converts a Smalltalk Character object into a C character.

Parameters

oeCharacter: Identifies the Character object to convert.

lpChar: Points to the location where the converted C character is placed.

Return Value

Returns the argument **oeCharacter**.

Comments

If oeOEToCchar fails, returns OEnonOop. If **oeCharacter** does not represent a valid handle, the error code is OEerrorNonOop. If **oeCharacter** is not a character, the error code is OEerrorWrongClass.

See Also

oeCopyCtoOEchar, oeIsCharacter

oeOEToCdouble

```
OEoop oeOEToCdouble(
 OEoop
   oeDouble,
 OEdouble
   *lpDouble);
```

The oeOEToCdouble function converts a Double object into a C double.

Parameters

oeDouble: Identifies the Double object to convert.

lpDouble: Points to the location where the converted double is placed.

Return Value

Returns the argument **oeDouble**.

Comments

If oeOEToCdouble fails, it returns OEnonOop. If **oeDouble** does not represent a valid handle, the error code is OEerrorNonOop. If **oeDouble** is not a double, the error code is OEerrorWrongClass.

See Also

oeCopyCtoOEdouble, oeIsDouble

oeOEToCfloat

```
OEoop oeOEToCfloat(
 OEoop
  oeFloat,
 OEfloat
  *lpFloat);
```

The `oeOEToCfloat` function converts a `Float` object into a C float.

Parameters

oeFloat: Identifies the `Float` object to convert.

lpFloat: Points to the location where the converted float is placed.

Return Value

Returns the argument **oeFloat**.

Comments

If `oeOEToCfloat` fails, returns `OEnonOop`. If **oeFloat** does not represent a valid handle, the error code is `OEerrorNonOop`. If **oeFloat** is not a float, the error code is `OEerrorWrongClass`.

See Also

`oeCopyCtoOEfloat`, `oeIsFloat`

oeOEToCint

```
OEoop oeToCint(
 OEoop
  oeInteger,
 OEint
  *lpInt);
```

The `oeOEToCint` function converts a `SmallInteger` object into a C integer.

Parameters

oeInteger: Identifies the `SmallInteger` object to convert.

lpInt: Points to the location where the converted integer is placed.

Return Value

Returns the argument **oeInteger**.

Comments

If `oeOEToCint` fails, returns `OEnonOop`. If **oeInteger** does not represent a valid handle, the error code is `OEerrorNonOop`. If **oeInteger** is not a `SmallInteger`, the error code is `OEerrorWrongClass`.

See Also

`oeCopyCtoOEint`, `oeIsInteger`

Unsafe Functions

The standard Object Engine interface is very defensive about problems in your C functions (and the Smalltalk code that calls them). Bounds-checking is performed on indices, class-checking is performed during coercion, and so on. However, such runtime checks have a negative performance impact.

For situations in which critical performance needs motivate you to bypass the safety mechanisms, you can compile your code using an unsafe version of the Object Engine access protocol.

It is not recommended that you do this in ordinary practice.

Even in a safe mode, your functions should perform consistency checks on passed arguments, at least checking for the correct class. When using the unsafe implementation, performing these checks is absolutely essential for correct operation of the system. Test your functions very carefully before compiling in an unsafe mode and using the resulting Object Engine on a useful image.

Caution: If you compile your functions for unsafe operation, they will run somewhat faster, but problems in your functions can crash the resulting Object Engine rather than fail. Worse yet, some problems can silently corrupt the virtual image (VI) without crashing. Finally, because hard-to-trace errors can be introduced so easily in unsafe mode, an unsafe Object Engine build and any VI's derived therefrom are not handled by Cincom technical support.

Specific Object Engine implementation details that are exposed by the unsafe interface code do not represent a supported Object

Engine interface. They change from release to release, and contain hidden dependencies and restrictions, which may render them useless outside this context. In other words, avoid using the unsafe Object Engine compilation option.

Appendix A

#define Operators

Topics

The following table indicates the operators that can be used in a `#define` expression.

Operators Permitted in #define Expressions

The operators are listed in descending order of precedence. The operators that appear on the same line have equal precedence.

Table 39: #define operators

Symbol	Type of Operation	Associativity
sizeof + - ~ !	Unary	Right to left
* / %	Multiplicative	Left to right
+ -	Additive	Left to right
<< >>	Bitwise shift	Left to right
< > <= >=	Relational	Left to right
== !=	Equality	Left to right
&	Bitwise AND	Left to right
^	Bitwise-exclusive OR	Left to right
|	Bitwise-inclusive OR	Left to right
&&	Logical-AND	Left to right
||	Logical-OR	Left to right
?:	Conditional	Right to left

Appendix B

Resolving Exceptions

Topics

Many common and difficult to resolve exceptions are caused by problems with external libraries. Some of these problems are unique to particular platforms and may be raised because of problems resolving the link between your External Interface classes and the shared library files. This appendix explains some of the more common errors that you may encounter and some strategies for resolving them.

Common ExternalInterface Exceptions

As we discussed earlier in this book, exceptions may be raised under four different general circumstances (see External Interface Exceptions). This appendix details the single most common variety of these four classes of exceptions, namely those that are raised when calling external libraries. A related issue — exceptions raised when using the Object Engine Access interface (for details on this interface, see Object Engine Access Functions) — ia also addressed later in this appendix.

Problems that occur when accessing external libraries may raise one of the three following exceptions:

- libraryNotFoundSignal
- libraryNotLoadedSignal
- externalObjectNotFoundSignal

The following sections of this appendix treat the possible causes of each of these three exceptions.

ExternalLibraryHolder>>libraryNotFoundSignal

This exception may be raised when the library or libraries specified in the libraryFiles attribute of your ExternalInterface class could not be located on the search path. You should check the libraryFiles and libraryDirectories attributes. If no path is specified, DLL and C Connect will look in the current working directory for the library, i.e. the directory where VisualWorks was started. If you make use of environment variables in your path specifier, you should be aware that the variables are set by the Object Engine once at startup time, and will not reflect any subsequent changes made in the operating system. This is a consequence of the way that most operating systems pass environment variables to processes.

ExternalLibrary>>libraryNotLoadedSignal

This exception is raised for a failure during the attempt to load the library that your ExternalInterface class indicates contains the target function or datum. If your exception handler does not catch this error, it will open a notifier with the error: Unhandled Exception: 'Unable to load library.' To resolve the cause of this exception, you can check for four common problems (note that the last two are platform-specific):

- Correct library type
- Interdependency
- Incorrect permission
- Incorrect version

1. First, make sure that the library you are trying to load is a shared library or a dynamically linked library and not a so-called “static” library. On some platforms, you may encounter libraries that require static linking (for example, on UNIX platforms this is indicated with the .so filename extension). These libraries will require that you add interface functions or re-link them as DLLs.
2. Libraries that depend on other libraries often cause problems. Make sure you are specifying all the libraries on which the library you are calling is dependent. You can do this by carefully setting all the interdependent libraries in the libraryFiles attribute in the definition of your ExternalInterface subclass. For details, see Incremental Loading of Dynamic-Link Libraries.

   On the MS-Windows platform, you can check to determine if this is the cause of the libraryNotLoadedSignal by opening up the debugger, and inspecting the errorCode object. If this SystemError object contains a parameter of 87, 126, or 1157 (ERROR_MODULE_NOT_FOUND), this strongly indicates a library interdependency problem.
3. A third common problem that results in the libraryNotLoadedSignal on UNIX platforms is when the library doesn’t have execute permissions set for you. Note that execute permissions are often stripped when FTPing a library, so make sure this is not causing the exception.

ExternalMethod>>externalObjectNotFoundSignal

The externalObjectNotFoundSignal simply means that VisualWorks located the DLL or shared library file, but was unable to find the desired function inside the library. Note that 16-bit function calls are no longer supported by Microsoft or VisualWorks, and the thunking DLLs are deprecated as of release 7.6.

To resolve the cause of this exception, you can check for three common problems:

- Failure to export the function
- Incorrect mapping of the C function prototype to the Smalltalk method definition

1. This exception is often raised because of a failure to export the function. This problem tends to occur on the MS-Windows platforms because the DLLs must have their functions exported to make them public (available to be called by a client).

   One way to export a function is to specify it in the export section of the `.def` file for the DLL. Older compilers also allow the use of `_export` in the function declaration to export that function. You should consult the documentation for your compiler to determine the proper way to export a function for a DLL. For details, see MS-Windows.

   Here is an example of using `_export` in a function declaration:

   ```
   int _export foo( int ) { return 1; }
   ```

2. The main types of C function “prototype mapping” errors are:

   - Failure to customize a 16-bit function prototype to make it available to a 32-bit client
   - Using the wrong calling convention

   You can check for these problems by performing the following steps:

   If you build a DLL using the C calling convention then your Smalltalk function definition will have to include the `_cdecl` type specifier, as follows:

   ```
   foo: arg1
     <C: _cdecl foo(short arg1)>
   ```

   You should ensure you are using the calling conventions used with the 32-bit implementation of Windows API. In the case where you have `Win32s` installed, there are 32-bit versions of the Windows API calls, as well as the 16-bit versions. For example, `kernel32.dll`, `gdi32.dll`, and `user32.dll` contain 32-bit implementations of what you might find in `kernel.dll`, `gdi.dll`, and `user.dll`.

The difference here is that some 32-bit versions of the Window's API calls have special considerations. For example, if the function accepts or returns a pointer that is going to hold some form of string (such as LPTSR, LPCTSR, etc.), then the function that you are going to call will have an 'A' appended to the end of it. For example, if you are trying to call the 32-bit implementation of GetProfileString() then you will need to call GetProfileStringA().

Consult your Microsoft SDK documentation for more information on how to use these 32-bit API calls. Of special interest for internationalization are those that end in 'W'.

Object Engine Access Interface Exceptions

While attempting to call functions in the Object Engine Access interface, you may encounter an externalObjectNotFoundSignal. This can occur when you load a DLL that calls the Object Engine, and you in turn try to call one of the functions in your DLL using your ExternalInterface class.

As mentioned in the chapter Platform Specific Information, dynamic-link/shared libraries that use the Object Engine access protocol must export the symbol oeLoadInitialize(). This function is called when the library is first loaded into the address space of the calling process to initialize local data used to implement the Object Engine protocol.

Exception Error Codes

If the name field in the SystemError object is either #'io error' or #'system error' then the errors were likely passed back from one of the called C functions. The following table summarizes some of the more common error codes you may encounter while using DLL and C

Connect on the MS-Windows platform. These codes are platform-specific, and comprise a subset of those found in the file `winerror.h`:

Table 40: Common Error Codes (MS-Windows)

If the value of _errorCode is:	The Error's Significance is:
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
8	ERROR_NOT_ENOUGH_MEMORY
11	ERROR_BAD_FORMAT
12	ERROR_INVALID_ACCESS
13	ERROR_INVALID_DATA
15	ERROR_INVALID_DRIVE
18	ERROR_NO_MORE_FILES
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER or ERROR_MOD_NOT_FOUND
110	ERROR_OPEN_FAILED
123	ERROR_INVALID_NAME
126	ERROR_MOD_NOT_FOUND
182	ERROR_INVALID_ORDINAL
190	ERROR_INVALID_MODULETYPE
193	ERROR_BAD_EXE_FORMAT
1157	ERROR_MOD_NOT_FOUND

Appendix

C

Examples

Topics

Using appropriate libraries, DLL and C Connect can be used to make calls directly from VisualWorks to APIs or external libraries in the host operating system. To illustrate this use of the product, this appendix presents two examples:

- How to use DLL and C Connect to allow VisualWorks to launch native applications on MS-Windows platforms.
- How DLL and C Connect can be used to improve the performance of vector arithmetic with an external library.

Launching Applications under Windows

VisualWorks provides a set of standard interfaces for calling APIs in a host OS. Platform-specific functionality is generally provided by subclasses of OSSystemSupport. For example, on MS-Windows platforms, subclasses of Win32SystemSupport provide protocol which allows a VisualWorks programmer to fork and terminate application processes in the Windows environment. DLL and C Connect is used to provide access to the functionality in Win32SystemSupport and its subclasses.

To illustrate the use of these interfaces to the host OS, this section explains how to make simple calls to the MS-Windows API.

For example, with DLL and C Connect, a VisualWorks programmer can call the MS-Windows API functions CreateProcessA() and TerminateProcess() in order to launch and quit Windows applications. Full documentation of this API (functions, parameters, and return values) can be found in the “Windows SDK Programmer’s Reference Volume 2: Functions.”

Following the architecture of DLL and C Connect, a subclass of ExternalInterface must be used that identifies the appropriate library and provides interface methods for the API functions.

The Win32SystemSupport Classes

VisualWorks uses the OSSystemSupport class hierarchy to provide a general framework for calling APIs in the host OS, and for MS-Windows, there are four specific subclasses which concretize the protocol: Win32SystemSupport, Win32sSystemSupport, and WinNTSystemSupport. For low-level API calls, you must use the class that corresponds to your platform.

Since the concrete subclasses of Win32SystemSupport inherit their behavior from their abstract superclass, these ExternalInterface classes should each have their virtual attribute enabled. Accordingly, to prepare the Win32SystemSupport classes for use, you must first set these attributes as follows:

1. Open a Hierarchy Browser on class Win32SystemSupport.

2. Set the beVirtual: parameter true in Win32SystemSupport and all of its subclasses. To do this, edit the definition text in the Browser template and then select accept on the <Operate> menu. Repeat this for every class below Win32SystemSupport.

For discussion of each parameter in these class definitions, see Defining External Methods. For the present example, you will only need to change the parameter to beVirtual: (note that these changes are only necessary in the present release of VisualWorks).

Launching an Application Process

Once the virtual attributes have been set, you may test the example by evaluating a Smalltalk expression like the following (the receiver should be the class appropriate for the platform you are using, i.e., WinNTSystemSupport):

```
WinNTSystemSupport CreateProcess: nil arguments: 'notepad'
```

A Windows Notepad application should launch on your display. If you get a notifier with the error "**External Object not found**", there has been some problem locating the external library. You should double check the parameters to the libraryFiles, libraryDirectories, and beVirtual attributes in all the Win32SystemSupport classes (for additional information on troubleshooting problems with loading DLLs, see Resolving Exceptions). For the Windows API, it is normally unnecessary to specify a path argument to the libraryDirectories attribute.

Launching and Terminating an Application Process

To launch and then terminate a Windows application, evaluate the following Smalltalk code fragment:

```
| p |
 p := WinNTSystemSupport
    CreateProcess: nil arguments: 'notepad'.
 (Delay forSeconds: 5) wait.
 ^WinNTSystemSupport TerminateProcess: p
```

Note that it is a common convention to use the actual C procedure name as the first keyword of the Smalltalk interface method.

Vector Functions

This section provides a simple example to illustrate how DLL and C Connect can be used to improve performance of computationally intensive applications. In this example, a vector functions package is implemented both as a Smalltalk class and as a C library. A common protocol is provided for both implementations, but for complex vector math the C library is used to increase performance.

The procedure for testing the example is as follows:

1. Build the library file using the native C development tools.
2. Load the parcel containing the vector math classes.
3. Configure the `ExternalInterface` class to call the external DLL.

For the purposes of this example, the commands used on MS-Windows NT 4.0 are shown.

Building the C Library

Following the steps outlined above, you must first build the vector library file on your platform. Complete source code for the library is provided in the `src` subdirectory of the DLL and C Connect installation. You should use the standard compiler for your platform; other compliers (i.e., `gcc`) may work, but the example code might need to be modified for proper compilation.

To simplify this step, a makefile has been provided in the `vector` subdirectory of the DLL and C Connect release. Choose the makefile that is appropriate for your platform, making a copy with the new name `makefile`. Additionally, it may be necessary to edit the definitions in the `makefile` so that the library can be generated without errors. To do this, check the following definitions against your installation of DLL and C Connect:

1. Open the `makefile` and ensure that the definition `SRC` refers to the `src` subdirectory in the DLL and C Connect installation (e.g., `SRC=..`

\src). All definitions are located at the beginning of the makefile text.

2. Ensure that the definition BRLIB refers to the nt subdirectory of the DLL and C Connect installation (e.g., BRLIB=..\nt). If you are using a different platform, check that this definition is appropriate.
3. Since the vector examples must be compiled with the standard release compiler for your platform, the definition MSDEV must point to the correct volume and installation directory for the Microsoft C Compiler (e.g., MSDEV=C:\MSDEV).
4. Save the edited makefile.

Next, use the makefile to generate the object file and the DLL. For example, under MS-Windows NT, open a command prompt window from the Start menu, connect to the directory dllcc\vector, and then issue the following command to execute the makefile:

```
nmake makefile
```

During compilation, some warnings may be generated about functions in vector.c, especially with regard to the conversion between double and float types. You may safely ignore these warnings —the library should still function properly.

Once the file vector.dll has been successfully generated, you can test the vector functions library from Smalltalk.

Calling the Vector Functions Package

Next, you must load the predefined vector math classes parcel into your VisualWorks image. To do this, open a File List window, and parcel in the file named DLLCCVectorMathExample.pcl (this file is located in the DLL and C Connect release directory). This parcel contains a small but complete vector math package, which is organized as three classes (Vector, VectorMathExternal, and VectorMathInternal). These classes should appear in the class category called Numeric-Vectors.

Organization of the Vector Math Library

Vector objects (instances of class Vector) are manipulated by vector the math functions, implemented either in Smalltalk or C. Class

VectorMathInternal provides a native Smalltalk implementation for vector math, while class VectorMathExternal contains calls to the external C functions. Note that VectorMathExternal is a subclass of ExternalInterface. Both of these vector math classes provide the same public interface for performing simple arithmetic operations, negation, rounding, cross product, and dot product.

Description of Class Vector

Instances of class Vector are n-element vectors of floating-point numbers, each stored as 4 bytes in the vector object. Elements in the vector are indexed using the method floatAt:. Thus, in a homogeneous 4-element vector, x is floatAt: 1, y floatAt: 5, z floatAt: 9, and an extra value (for example, alpha) is floatAt: 13.

Class Vector includes one class variable VectorLibrary which references the library that is used for mathematical operations.

Configuring Class VectorMathExternal

The class VectorMathExternal must be configured to reference the library file vector.dll. All the standard names for this library (vector.dll vector.sl vector.so) are already defined in this class, but you will need to also define the libraryDirectories attribute so that the DLL can be located. The simplest way to do this is to open a Browser on class VectorMathExternal, and then define libraryDirectories as the path to the DLL; for example:

```
#(#libraryDirectories #('C:\vw7\dllcc\vector '))
```

Testing the External Vector Math Library

Once the DLL has been built, the Smalltalk classes have been loaded and configured, you are ready to test the installation. In class Vector, the category named **testing** contains two methods which you may use to check that the external vector library has been properly installed.

Open a Workspace window and try evaluating the following Smalltalk expression (select print it from the <Operate> menu):

```
Vector test1
```

This method performs a number of vector operations, and compares the results of the Smalltalk and C implementations. If the external vector library has been compiled and loaded properly, the result of the method should be an empty `Dictionary` object.

If the resulting `Dictionary` is not empty, the results returned by the internal and external packages did not match. This normally indicates that the external library was not loaded properly, and that the external calls failed. If you encounter this error, you should check the definition for class `VectorMathExternal` and make sure that the path stored in the `libraryDirectories` attribute is correct. For additional information on troubleshooting problems with DLLs, see Resolving Exceptions.

A second test method allows you to measure the performance of the external library. Evaluate the following Smalltalk expression:

```
Vector test2
```

This method will return an `Association`, whose key contains the time (in milliseconds) required to run a computational test for both the internal and the external vector functions. The value of the association contains the (negative) percentage increase of speed between the two vector implementations.

To test the cross-product function, evaluate the following Smalltalk expression in a Workspace:

```
1 , 2 , 3 crossProduct: 1 , 1 , 1
```

The result should be a vector: `(-1.0, 2.0, -1.0)`.

By browsing class `Vector`, you can find a number of additional sample methods in the protocol named examples.

Appendix

D

Limitations

Topics

DLLCC has several limitations that may impact an application’s performance and behavior significantly. Moreover, the causes are often invisible because they originate in source code or executable binaries that are not immediately evident from within a Smalltalk development environment. This section examines a few of these shortcomings in detail.

Error Checking

When accessing operating system APIs it is in many cases important to use the _wincall or _syscall qualifiers on the call. This is because these may rely on global state such as errno (Unix) or the windows equivalent accessed via GetLastError(). The errors should be retrieved before any other API call is made which could overwrite the error value. Because we cannot guarantee that at the image level (for example, errors due to interactions with the VM or Smalltalk process switches), the _wincall or _syscall modifiers cause the VM to do so. However, this does not cover all possible situations of this nature. For example, the Windows sockets API provides its own special global error code, accessed via WSAGetLastError(). This is not covered by such a modifier, and thus it may not be possible to call Windows Socket APIs via DLLCC and reliably get the error code.

Consider, for example, a hypothetical Windows API providing a function called Foo(). This function is documented to return zero when it fails. The corresponding MSDN documentation states that, when Foo() fails, the program should call GetLastError() to find out what caused the problem. At first glance, the following method seems plausible:

```
Win32SystemSupport>>foo
 | result error |
 result := self actuallyCallFoo.
 result = 0 ifFalse: [^result].
 error := self actuallyCallGetLastError.
 ^self fooFailedWith: error
```

However, a process switch may occur between the invocation of actuallyCallFoo and actuallyCallGetLastError. If a Smalltalk process switch occurs, the new process may also call a function that sets or resets the last error retrieved by GetLastError(). By the time another process switch occurs and GetLastError() is called in the method foo, the last error retrieved may no longer correspond to the function Foo().

At first, it may seem that the above foo method should be atomic with respect to process switches, so one could wrap the method body in a block which receives the message valueUnpreemptively. This approach is extremely brittle because it comes with serious potential side effects. Nevertheless, it is just as unsafe as the original foo implementation because the VM may also call `GetLastError()` (or reset the last error value) on its own. Consider the case in which, when coming back from the DLLCC primitive that invokes `Foo()`, the VM decides to check incoming signals from the operating system. Consequently, the VM may examine these events, and the functions invoked to do so may very well overwrite the last error with zero if no error occurs, or with any other value if a new error occurs on top of that which happened as a result of `Foo()`.

For the cases in which calling `GetLastError()` must be called after a Windows API call, DLLCC provides the `_wincall` function declaration keyword. In this way, if the C pragma in `Foo()` includes the keyword `_wincall`, then if `Foo()` answers zero, then the primitive that invokes `Foo()` will fail with an error code corresponding to the answer to `GetLastError()`. Note that this calling convention ensures that the VM will treat the invocation of `Foo()` and `GetLastError()` atomically, without any other Windows API function calls in between. This approach guarantees that the call to `GetLastError()` performed by the VM will return the right error code after calling `Foo()`.

In general, function calls and their corresponding error handling must be atomic with respect to both the image and the VM. Although DLLCC also provides `_syscall` (which returns `errno()` instead of `GetLastError()`), the approach does not scale to every API. For instance, Windows sockets API uses `WSAGetLastError()` instead of `GetLastError()`. Consequently, if a keyword such as `_winwsacall` is not available, then in general it is not safe to implement image methods such as the following because Smalltalk process switches must be accounted for:

```
Win32SystemSupport>>wsaResetEvent
 | result error |
 result := self actuallyCallWSAResetEvent.
 result = 0 ifFalse: [^result].
 error := self actuallyCallWSAGetLastError.
 ^self wsaResetEventFailedWith: error
```

Even if those were avoided via `valueUnpreemptively` or perhaps mutex protection, the VM itself may invoke the Windows sockets API between the call to `WSAResetEvent()` and `WSAGetLastError()`. Note how, in this case, the VM behavior that is critically important to the application developer is also invisible from the image. Consequently, particularly for operating system calls, the interface must be implemented with extreme care to ensure that Smalltalk process switches and VM activity do not conflict with the image's use of DLLCC.

To some extent, the image's communication with C may be regarded as a thread in a multithreaded application, where at least one thread is in the image and at least one thread may be in the VM. If critical sections cannot be protected from the image, then you should consider one of the following strategies:

- Write a DLL that provides atomic error checking and critical section protection. While the VM is executing the primitive that synchronously calls the DLL, it will not execute other Smalltalk or object engine code. Consequently, if the DLL provides atomic error protection, chances are this will be enough to ensure proper application behavior.

  Note, however, that some VMs have multiple execution threads. The possibility that threads other than the object engine thread could conflict with DLLCC when interfacing certain APIs must also be accounted for.
- Write a primitive in the VM and invoke it. This has the same effect as writing a DLL, with the exception that the linkage to the wrapped API is done in terms of indexed primitives as opposed to via DLLCC. As a result, you will have to compile a VM.
- Write a user/named primitive and invoke it. This has the same effect as writing a DLL, with the exception that the linkage to the wrapped API is done in terms of user/named primitives as opposed to via DLLCC. You will have to compile a DLL just as with the DLLCC approach.

Note that if you choose to write primitives, you will have more control over argument marshalling. This is because the primitives may assume that the receivers of the user/named primitives have a particular object structure that may facilitate argument marshalling.

Note also that the above arguments about GetLastError() also apply to errno. Never write code such as:

```
UnixSystemSupport>>errnoLocation
 <C: int * __errno_location()>
 ^self externalAccessFailedWith: _errorCode
```

This code example shows warning signs that should be heeded by the developer. There is the standard POSIX definition of errno which states that errno is simply something that can be assigned to, and that it has type int. Nowhere do the relevant specifications state exactly how errno must be implemented. One could go through the relevant .h files and determine the actual errno location for a specific platform, but doing so is absolutely not portable even across slightly different versions of the same operating system. In general, examining .h files to copy private implementation details into a DLLCC external interface is a bad idea. Moreover, relying on anything that begins with an underscore (or, even worse, two underscores) should be seen with a strong dose of skepticism.

Special care must be taken to select and implement one of the above strategies. For example, consider the need to use the function dlerror(). The VM already uses dlerror() in various primitives. Since the VM knows there is only one thread calling dlerror(), the VM does not need to take into account that dlerror() may not be thread safe (as per the Single Unix Specification document specifying the behavior of dlerror()). Hence, user code must make calls to dlerror() from the VM thread as well. In other words, the specific details of how dlerror() is implemented limit the strategies described above. For instance, if dlerror() is called from a user DLL, calls to the DLL must not be threaded. Similarly, such user DLLs must not create threads that subsequently call dlerror().

Constructs that Cannot be represented in DLLCC

The C compilation environment includes several entities that cannot be represented faithfully in a DLLCC environment. This limitation can cause problems while parsing some .h files, or determining which declarations to include in external interface classes.

For instance, C compilers typically provide private (and not so private) compiler macros that identify the compiler in use. For the sake of illustration, GCC defines several macros including `__GNUC__`. Sometimes, `.h` files supplied with operating systems will include conditional declarations that resemble the following:

```
#if __GNUC__
 /* then do something specific for GCC */
#else
 /* then do something generic */
#endif
```

Depending on the conditionalized statements, it may or may not be possible to provide an accurate DLLCC rendering of the `.h` file in question. In theory, one could argue that the external interface being implemented could also provide a declaration for `__GNUC__`, if the necessary investigation determines it is a suitable course of action. However, the problem with this approach is that it violates the fact that some of the compiler-declared macros are private. Therefore, implementing such macros in external interface classes may necessitate significant maintenance and manual work. For instance, some Windows compilers define `WIN32`, but all compilers should define `_WIN32`. Should DLLCC declare `WIN32`, `_WIN32`, or both? Depending on the `.h` file being parsed, this decision may not be trivial.

Compilers may also inline certain functions such as `memcpy()` instead of calling the corresponding C library implementation, called "intrinsic functions." If DLLCC calls a function that the C compiler for the platform in question implements as an intrinsic function, the behavior invoked by DLLCC will not be the same as that provided by a C compiler. Something similar occurs with linker macros.

These differences may expose programs to implementation variations that should be considered. For example, according to the relevant documentation, Linux kernels prior to 2.6 do not have direct support for POSIX threads. On kernels 2.4 and below, glibc provides its own thread implementation called LinuxThreads. A modern glibc application running on a 2.6 kernel will use POSIX threads, or LinuxThreads if the kernel is 2.4 and below. This is transparent to the application, except that the different threading mechanisms use a different number of realtime signals for their implementation purposes. Consequently, it is not safe to assume your application will have access to the 32 realtime signals specified by the POSIX specification. Instead, Linux applications have access to 30 or 29 realtime signals, depending on whether glibc uses POSIX threads or LinuxThreads. In addition to these divergences from the POSIX specification, Linux does things differently because it can specify a different number of realtime signals available to applications. The specification states very clearly that this variability must be accounted for, and that applications must have code that verifies they have access to enough realtime signals. However, the realtime signal counts that must be compared may be defined in terms of macros the values of which change at runtime.

Data Type Size and Alignment Variation

In general, it is not safe for developers to assume types such as `int` have a definite size. Using `sizeof()`, however, is problematic with DLLCC. For instance, the C99 specification states that the size of an `int` is between those of a short and a long, but it does not specify an explicit int size. Therefore, C programmers should implement their programs using `sizeof(int)`. Note that, although `sizeof()` looks like a function call, it is a compiler-provided operator. Consequently, since DLLCC does not provide a complete C compilation environment, you cannot use DLLCC to call `sizeof()`.

This limitation has potentially dangerous consequences that go beyond the size of elementary types. Consider a struct consisting of 2 ints:

```
struct {int a; int b;} ints;
```

At first sight, it may be tempting to declare this struct in DLLCC as is. Most likely, such a declaration will result in a struct that requires 8 bytes of storage. However, at least the Sun C compilers provide switches that control the alignment of data types in memory. Therefore, depending on the compiler switches used to compile a program using the above struct, `sizeof(struct ints)` may return 8, but it may also return 12 if the compiler is told to align 4 byte entities on 8 byte boundaries, and 16 if the compiler is additionally told to align longer entities on 16 byte boundaries. Therefore, if DLLCC is used to interface a DLL compiled with particular compiler switches, it may not be enough to simply parse a `.h` file and import the structure declarations. In addition to this, the compilation environment must be closely observed so that the resulting interface will match the expected invocation and data structure characteristics.

Also, note that the compiler switches just mentioned also control which instructions will be used to read and write memory. Consider the following struct:

```
struct {byte a; double b;} misaligned;
```

The SPARC processors will typically use an LDDF instruction to fetch the field b in the above struct. Compilers typically assume that doubles are always aligned to 8 byte boundaries and, therefore, the above struct will require 16 bytes of storage. However, that is not necessarily the case. One could define the above struct without any padding so that it requires 9 bytes of storage. How will a C program react to this variation?

On x86 processors, the corresponding Intel manuals clearly state that loading a misaligned double will cause severe performance penalties. Nevertheless, the load will succeed regardless of the alignment. Therefore, C compilers may issue FLD instructions without worrying about the address of the double to load. Whether or not the double is aligned may be left to the application developer. On SPARC processors, on the other hand, the corresponding SPARC specification states that whether the LDDF instruction succeeds to load a misaligned double or not is implementation defined. In other words, it can be the case that different SPARC chips produced by different manufacturers provide a different implementation for the LDDF instruction even if the chips implement the same SPARC specification. There are various ways in which an LDDF instruction failure will be handled. Thus, a DLLCC program that passes a pointer to the double member of the misaligned struct may or may not cause an external function to crash depending on the SPARC CPU executing the function call, the compiler switches used to compile the external function in question, and the specific manner in which the system will react to an LDDF failure.

As a side note, note that GCC requires 8 byte alignment for doubles by default. Moreover, although GCC does provide a switch to cause the generated code to assume some doubles are not 8 byte aligned, GCC does not provide a switch to schedule instructions that will assume every double is misaligned. In their most general form, it is very difficult to impossible to capture these variations with DLLCC.

In any case, VisualWorks VMs do not currently provide a SIGBUS handler for the SPARC platform because the VM implementation does not require such handlers. Moreover, because DLLCC cannot assume the alignment of certain data types, entities such as doubles and long long integers will be accessed on a byte-by-byte basis to ensure that no SIGBUS exceptions will occur. Finally, it may not be appropriate to handle SIGBUS exceptions coming from third party functions.

These alignment problems may also occur when allocating uninterpreted bytes in fixed space. Consider evaluating the following code in a 32-bit image:

```
| u |
u := UninterpretedBytes newInFixedSpace: 16.
u doubleAt: 1 put: Double pi
```

At this point, if the first byte of a pointer to `u` is passed to an external function so `Double pi` can be accessed, a `SIGBUS` exception may occur because object bodies are aligned to 4 byte boundaries. And, while a 64-bit image imposes 8-byte alignment boundaries for object bodies, different data types may require even larger alignment boundaries. For the sake of illustration, some Windows file I/O APIs require cluster-size alignment boundaries. Consequently, fixed space allocations may not always be the best choice when using DLLCC.

Interface Limitations to Operating System APIs

In addition to these problems, operating system APIs governed by the Single Unix Specification can be particularly troublesome to interface using DLLCC. For example, the Single Unix Specification interface definitions for sockets and timers clearly state that the functions provided by the API may be implemented as functions or as macros. In other words, the APIs are sometimes defined in terms of mere identifiers. Developers are not even supposed to know whether the API calls written in C will resolve as actual function calls or as chunks of code provided by macros. This is reasonable because this implementation difference will be encapsulated and made invisible by the C compilation environment.

For the purposes of DLLCC, if the compilation environment does not provide a macro for the API interface identifiers, then calling the API via DLLCC is acceptable. If, however, the compilation environment provides both a macro and a function implementation for the API interface identifiers, then calling the function from DLLCC is not acceptable because it is a violation of the API's encapsulation. Even worse, if the compilation environment only provides a macro implementation, then it becomes clear that using the API according to its specification is impossible because DLLCC cannot call macros.

It is still possible to interpret the macros and manually reimplement them in DLLCC to the point that some actual external function can be called. However, application developers are not supposed to know these implementation details. Consequently, reifying private implementation details in DLLCC constitutes an implementation encapsulation violation of the corresponding API. Therefore, unlike an equivalent, standard-compliant C program, the resulting DLLCC code will not be portable across all the platforms covered by standards such as the Single Unix Specification. Even more importantly, the consequences of violating the specification of a C API are typically described in terms of undefined behavior. The term "undefined behavior" might mean something like "segmentation fault" in 99% or so of the cases. Note well, however, that it is also possible for an application exhibiting undefined behavior to apparently continue working for a significant amount of time while silently corrupting data. Hence, application developers interfacing C APIs must pay very close attention to all occurrences of the term "undefined behavior", and the conditions in which undefined behavior may occur.

DLLCC Does Not Support Specification-based Encapsulation

As detailed in the above discussion, C specifications may define an API in terms of functions or macros. In addition, the relevant data types required by the API may be specified incompletely as well. Consider the timer API defined by the Single Unix Specification. According to the Single Unix Specification, the struct called sigevent can have some of its fields defined inside a union. Therefore, the specification states developers should only access a certain portion of the sigevent struct to avoid inadvertently corrupting private implementation data. Moreover, the Single Unix Specification does not list all the struct fields, and does not specify the order in which the fields appear in the struct either. This means that different sigevent structs may have different layouts and sizes, and it may not be appropriate to define the struct sigevent in a single class such as UnixSystemSupport.

This situation is problematic for DLLCC because the specifications allow different operating systems, or even different distributions of the same operating system, to provide their own definition for the sigevent structure. In theory, users would download the source code and compile it against their operating system. Thus, although these data type differences are exposed at the API level because functions or macros take sigevent struct arguments, a C compilation environment will render data type variations invisible for C applications. This type of encapsulation has far-reaching consequences. For example, in DLLCC, developers will have to specify what the private parts of sigevent would look like if they ever used an API that references sigevent from the image. Imagine the following hypothetical scenario:

```
UnixSystemSupport>>timer_create: clockid with: evp with: timerid
 <C: int _syscall timer_create(clockid_t clockid, struct sigevent *evp,
  timer_t *timerid)>

 ^self externalAccessFailedWith: _errorCode
```

The definition of the sigevent struct might look like this:

```
UnixSystemSupport>>sigevent
 <C: struct linux_sigevent { sigval_t sigev_value;
int sigev_signo;
int sigev_notify;
void (*_function)(sigval_t);
void *_attribute; }>
```

Although the Linux programmer manual states there may be unions in this struct, there is no reference to unions in the DLLCC code. Consequently, it would be reasonable to conclude the above code is generally unsafe and bound to fail. Further examination reveals additional problems. For example, the relevant `.h` files that define the `sigevent` struct show that the Linux structs are padded to 64 bytes on both 32- and 64-bit x86 Linux, and on 32-bit PowerPC Linux. Since the various paddings are not reified in the DLLCC specification, we should conclude the above code is unsafe. For the sake of completeness, Solaris pads `sigevent` to 24-bytes on 32-bit SPARC and 32-bit x86, and to 40 bytes on 64-bit SPARC and 64-bit x86. Meanwhile AIX 6.1 and OS X 10.4 do not pad.

To summarize, the sizes of the `sigevent` struct are as follows:

Platform	**Struct Size**
64-bit Linux on x86, 32-bit Linux on x86, and PowerPC	64 bytes
64-bit Solaris on SPARC and x86	40 bytes
32-bit Solaris on SPARC and x86	24 bytes
32-bit Mac OS X on x86 and PowerPC, AIX 6.1	20 bytes

No single DLLCC definition can be simultaneously correct for all the above cases. Keep in mind that, even if the struct has the same size on different operating systems or operating system distributions, the struct fields may be different in any case. When interfacing this type of API, it is strongly recommended to write the interface in C (numbered or user/named primitives, or a wrapped API call in a DLL).

The same argument applies to other POSIX APIs such as those of sockets, and even file system related APIs such as `stat()`. Consequently, DLLCC should not be used to call such APIs because the necessary private implementation details are not representable in DLLCC.

Index