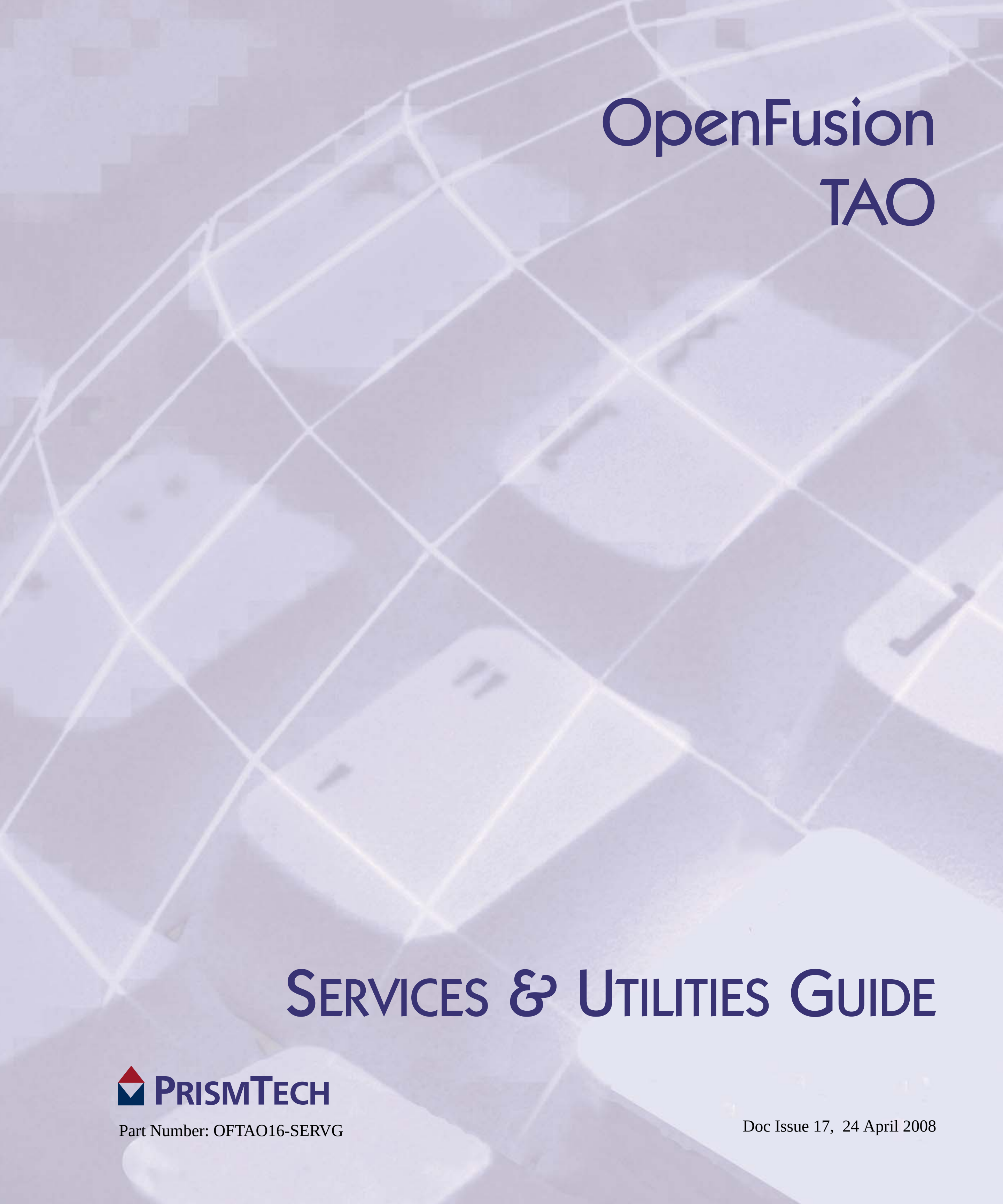
OpenFusion® TAO

Version 1.6

Services & Utilities Guide

PrismTech



OpenFusion TAO

Services & Utilities Guide

PrismTech

Part Number: OFTAO16-SERVG

Doc Issue 17, 24 April 2008

Copyright Notice

The information contained in this document is subject to change without notice and is made available in good faith without liability on the part of PrismTech Limited or PrismTech Corporation.

CONTENTS

Table of Contents

Services & Utilities

Preface

About the Services & Utilities Guide

The *Services & Utilities Guide* is included with OpenFusion TAO. The guide describes how to use the command line tools, services, and utilities, including the IDL compiler, provided with OpenFusion TAO.

Intended Audience

The *Services & Utilities Guide* is intended to be used by developers or those who need to use the services and utilities provided with OpenFusion TAO.

Organisation

The *Services & Utilities Guide* is divided into the following sections:

- Chapter 1, *TAO IDL Compiler*, describes how to use the OpenFusion TAO IDL compiler
- Chapter 2, *Interface Repository Service*, describes the Interface Respository
- Chapter 3, *Naming Service*, describes how to run the Naming Service from the command line, as well as describing utilities which can be used to manage the service
- Chapter 4, *Event Service*, describes how to run the Event Service from the command line, as well as describing how to manage the service
- Chapter 5, *Utilities*, describes useful command line tools

Conventions

The conventions listed below are used to guide and assist the reader in understanding the Services & Utilities Guide.

⚠	Item of special significance or where caution needs to be taken.
i	Item contains helpful hint or special information.
WIN	Information applies to Windows (e.g. NT, 2000, XP) only.
UNIX	Information applies to Unix based systems (e.g. Solaris) only.
C	C language specific
C++	C++ language specific
Java	Java language specific

Hypertext links are shown as *blue italic underlined.*

On-Line (PDF) versions of this document: Items shown as cross references, e.g. *Contacts* on page viii, are as hypertext links: click on the reference to go to the item.

```
%  Commands or input which the user enters on the
   command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, ““);
```

Italics and ***Italic Bold*** indicate new terms, or emphasise an item.

Arial Bold indicate Graphical User Interface (GUI) elements and commands, for example, **File | Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

Corporate Headquarters

PrismTech Corporation
6 Lincoln Knoll Lane
Suite 100
Burlington, MA
01803
USA

Tel: +1 781 270 1177
Fax: +1 781 238 1700

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web:	*http://www.prismtech.com*
General Enquiries:	*info@prismtech.com*

Services & Utilities

CHAPTER 1 TAO IDL Compiler

1.1 Introduction

This section describes the TAO IDL compiler's options and features. Users should be familiar with standard OMG IDL before reading this section or using the TAO IDL compiler. For information on the OMG IDL please refer to the IDL documentation provided on the OMG's web site (*http://www.omg.org/*).

1.1.1 Running

The TAO IDL compiler is run from the command line using:

```
%  tao_idl [options]
```

where *[options]* are zero or more of the command line options described under Section 1.6, *Compiler Options*, on page 5.

1.2 Generated Files

The IDL compiler generates a number of files from each *.idl* file. The generated file names are obtained by taking the IDL *basename* and appending a letter to the basename which signifies if the file is for a stub, skeleton, or skeleton template, then appending an extension for its file type (interface (.i), header (.h), or definition (.cpp). The complier provides options which enable different suffixes to be generated if required:

- client stubs - **C.i*, **C.h*, and **C.cpp*
- server skeletons - **S.i*, **S.h*, and **S.cpp*
- server skeleton templates - **S_T.i*, **S_T.h*, and **S_T.cpp*

TAO's IDL compiler creates separate **.i* and **S_T.** files to improve the performance of the generated code. Note that only the client stubs declared in the **C.h* file and the skeletons in the **S.h* file need to be *#include*d in your code.

1.3 Environment Variables

Table 1 Environment Variable Descriptions

Variable	Usage
TAO_IDL_PREPROCESSOR	Used to override the program name of the preprocessor that the TAO IDL compiler (*tao_idl*) uses.
TAO_IDL_PREPROCESSOR_ARGS	Used to override the flags passed to the preprocessor that *tao_idl* uses. This can be used to alter the default options for the preprocessor and specify things like include directories and how the preprocessor is invoked. Two flags that will always be passed to the preprocessor are -DIDL and -I.
TAO_ROOT	Used to determine where orb.idl is located.
ACE_ROOT	Used to determine where orb.idl is located.

Because the TAO IDL compiler does not contain code to implement a preprocessor, it must use an external one. For convenience, it uses a built-in name for an external preprocessor to call. During compilation, this is how that default is set:

1. If the macro *TAO_IDL_PREPROCESSOR* is defined, then it will use that.
2. Else if the macro *ACE_CC_PREPROCESSOR* is defined, then it will use that.
3. Otherwise, it will use "*cc*"

The same behaviour occurs for the *TAO_IDL_PREPROCESSOR_ARGS* and *ACE_CC_PREPROCESSOR_ARGS* macros.

Case 1 is used by the *Makefile* on most machines to specify the preprocessor.

Case 2 is used on Windows and platforms that need special arguments passed to the preprocessor (MVS, HPUX, etc.).

Case 3 is not normally used, but is included as a default case.

Since the default preprocessor may not always work when tao_idl is moved to another machine or used in cross-compilation, it can be overridden at runtime by setting the environment variables TAO_IDL_PREPROCESSOR and TAO_IDL_PREPROCESSOR_ARGS.

In previous versions, the environment variables *CPP_LOCATION* and *TAO_IDL_DEFAULT_CPP_FLAGS* were used for this purpose. Both will still work, but tao_idl will display a deprecation warning if it detects them. It is possible that support for these variables will be removed in a future version of TAO.

If *TAO_ROOT* is defined, then `tao_idl` will use it to include the *$(TAO_ROOT)/tao* directory. This is to allow `tao_idl` to automatically find *<orb.idl>* when it is included in an IDL file. `tao_idl` will display a warning message when neither is defined.

1.4 Operation Demuxing Strategies

The server skeleton can use different demuxing strategies to match the incoming operation with the correct operation at the servant. TAO's IDL compiler supports perfect hashing, binary search, and dynamic hashing demuxing strategies. By default, TAO's IDL compiler tries to generate perfect hash functions, which is generally the most efficient and predictable operation demuxing technique. To generate perfect hash functions, TAO's IDL compiler uses *gperf*, a general-purpose perfect hash function generator.

If you cannot use perfect hashing, then the next best operation demuxing strategy is using binary search, which can be configured using TAO's IDL compiler options (see Section 1.6, *Compiler Options*, below).

1.5 Collocation Strategies

`tao_idl` can generate collocated stubs using two different collocation strategies. It also allows you to suppress and enable the generation of the stubs of a particular strategy. You can generate stubs for both collocation strategies (using both *-Gp* and *-Gd* flags at the same time) and defer the determination of collocation strategy until run time. However, if you want to minimize the footprint of your program, then you might want to pre-determine the collocation strategy you want and only generate the right collocated stubs (or not generate any at all using both *-Sp* and *-Sd* flags at the same time, provided it's a pure client.)

1.6 Compiler Options

TAO's IDL compiler invokes your C or C++ preprocessor to resolve included IDL files. It takes the common options for preprocessors (such as *-D* or *-I*). The compiler also takes other options that are specific to it. Table 2, *Compiler Options*, shown following, describes each compiler option.

Table 2 Compiler Options

Option	Description	Remarks
-u	The compiler prints out the options that are given below and exits clean	
-V	The compiler printouts its version and exits	
-Wb,option_list	Pass options to the TAO IDL compiler, as follows:	
	skel_export_macro=macro_name	The compiler will emit macro_name right after each class or extern keyword in the generated skeleton code (*S* files,) this is needed for Windows that requires special directives to export symbols from DLLs, usually the definition is just a space on unix platforms.
	skel_export_include=include_path	The compiler will generate code to include include_path at the top of the generated server header, this is usually a good place to define the server side export macro on Windows.
	stub_export_macro=macro_name	The compiler will emit macro_name right after each class or extern keyword in the generated stub code: this is needed for Windows that requires special directives to export symbols from DLLs, usually the definition is just a space on unix platforms.
	stub_export_include=include_path	The compiler will generate code to include include_path at the top of the client header, this is usually a good place to define the export macro.
	export_macro=macro_name	This option has the same effect as issuing *-Wb*, *skel_export_macro=macro_name* *-Wb*, *stub_export_macro=macro_name*. This option is useful when building a DLL containing both stubs and skeletons.

Table 2 Compiler Options (Continued)

Option	Description	Remarks
-Wb,option_list (continued)	export_include=include_path	This option has the same effect as specifying *-Wb*, *stub_export_include=include_path*. This option goes with the previous option to build DLL containing both stubs and skeletons.
	pch_include=include_path	The compiler will generate code to include *include_path* at the top of all TAO IDL compiler generated files. This can be used with a pre-compiled header mechanism, such as those provided by Borland C++ Builder or MSVC++.
	obv_opt_accessor	The IDL compiler will generate code to optimise access to base class data for value types.
	pre_include=include_path	The compiler will generate code to include *include_path* at the top of the each header file, before any other include statements. For example, *ace/pre.h*, which declares compiler options for the Borland C++ Builder and MSVC++ compilers, is included in this manner in all IDL-generated files in the TAO libraries and CORBA services.
	post_include=include_path	The compiler will generate code to include *include_path* at the bottom of the each header file. For example, *ace/post.h*, which restores compiler options for the Borland C++ Builder and MSVC++ compilers, is included in this manner in all IDL-generated files in the TAO libraries and CORBA services.
-E	Only invoke the preprocessor	
-d	Causes output of a dump of the AST	
-Dmacro_definition	Passed to the preprocessor	
-Umacro_name	Passed to the preprocessor	
-Iinclude_path	Passed to the preprocessor	

Table 2 Compiler Options (Continued)

Option	Description	Remarks
`-Aassertion`	Passed to the preprocessor	
`-Yp, path`	Specifies the path for the C preprocessor	
`-H, option_list`	Pass options to the TAO IDL compiler, as follows:	
	`perfect_hash`	Generate skeleton code that uses perfect hashed operation demuxing strategy, which is the default strategy. Perfect hashing uses *`gperf`* program, to generate demuxing methods.
	`dynamic_hash`	Generate skeleton code that uses dynamic hashed operation demuxing strategy.
	`binary_search`	Generate skeleton code that uses binary search based operation demuxing strategy.
	`linear_search`	Generate skeleton code that uses linear search based operation demuxing strategy. Note that this option is for testing purposes only and should not be used for production code since it's inefficient.
`-in`	To generate *`#include`* statements with <>'s for the standard include files (e.g. *`tao/corba.h`*) indicating them as non-changing files	
`-ic`	To generate *`#include`* statements with `""`s for changing standard include files, (e.g. *`tao/corba.h`*).	
`-g`	To specify the path for the perfect hashing program (*`gperf`*). The default is *`$TAO_ROOT/bin/gperf`*.	

Table 2 Compiler Options (Continued)

Option	Description	Remarks
`-o`	To specify the output directory to IDL compiler as to where all the IDL-compiler-generated files are to be put. By default, all the files are put in the current directory from where is called.	
`-hc`	Client's header file name ending. Default is "*C.h*".	
`-hs`	Server's header file name ending. Default is "*S.h*".	
`-hT`	Server's template header file name ending. Default is "*S_T.h*".	
`-cs`	Client stub's file name ending. Default is "*C.cpp*".	
`-ci`	Client inline file name ending. Default is "*C.i*".	
`-ss`	Server skeleton file name ending. Default is "*S.cpp*".	
`-sT`	Server template skeleton file name ending. Default is "*S_T.cpp*".	
`-si`	Server inline skeleton file name ending. Default is "*S.i*".	
`-st`	Server's template inline file name ending. Default is "*S_T.i*".	
`-t`	Temporary directory to be used by the IDL compiler.	UNIX: use the `TEMPDIR` environment variable if defined, else use `/tmp/`. Windows: use TMP or TEMP environment variables, if defined, else use the Windows directory.
`-Cw`	Output a warning if two identifiers in the same scope differ in spelling only by case (default is the output of error message).	This option has been added as a nicety for dealing with legacy IDL files, written when the CORBA rules for name resolution were not as stringent.

Table 2 Compiler Options (Continued)

Option	Description	Remarks
`-Ce`	Output an error if two indentifiers in the same scope differ in spelling only by case (default).	
`-GC`	Generate AMI stubs ("*sendc_*" methods, reply handler stubs, etc)	
`-Ge flag`	If the value of the flag is *0*, *tao_idl* will generate code that will use native C++ exceptions. If the value of the flag is *1*, `tao_idl` will generate code that will use the *CORBA::Environment* variable for passing exceptions. If the value of the flag is *2*, the C++ *throw* keyword will be used in place of *ACE_THROW_SPEC*, *ACE_THROW*, and *ACE_RETRHOW* (*ACE_THROW_RETURN* and *TAO_INTERCEPTOR_THROW* will still be used).	
`-Gp`	Generated collocated stubs that use *Thru_POA* collocation strategy (default)	
`-Gd`	Generated collocated stubs that use Direct collocation strategy	
`-Gsp`	Generate client smart proxies	
`-Gt`	Generate optimised TypeCodes	
`-Gv`	Generate code that supports Object-by-Value	
`-GI`	Generate templates files for the servant implementation	
`-GIh arg`	Servant implementation header file name ending	
`-GIs arg`	Servant implementation skeleton file name ending	

Table 2 Compiler Options (Continued)

Option	Description	Remarks
`-GIb arg`	Prefix to the implementation class names	
`-GIe arg`	Suffix to the implementation class names	
`-GIc`	Generate copy constructors in the servant implementation template files	
`-GIa`	Generate assignment operators in the servant implementation template files	
`-Sa`	Suppress generation of the Any operators	
`-Sp`	Suppress generation of collocated stubs that use *`Thru_POA`* collocation strategy	
`-Sd`	Suppress generation of collocated stubs that use Direct collocation strategy (default)	
`-St`	Suppress generation of the TypeCodes	Also suppresses the generation of the *`Any`* operators, since the *`Any`* `>>=` operator needs the associated typecode.
`-Sc`	Suppress generation of the tie classes, and the *`*S_T.*`* files that contain them.	
`-Sv`	Suppress value type support (default).	

CHAPTER 2 Interface Repository Service

2.1 Running the Service

The Interface Repository makes all IDL declarations available.

2.1.1 IFR_Service

To run the Interface Repository you need to use the executable IFR_Service. This is found in the bin directory of the OpenFusion TAO distribution.

Table 3 Interface Repository Command Line Options

Option	Description
`-a <base_address>`	This option only works when you also specify the *-p* option. The *-a* option gives the base address for memory mapped persistence and may be specified in decimal, octal (with a leading zero) or hexadecimal (with a leading 0x). If this value is not specified and the Interface Repository is made persistent by using the `-p` option, then this base address defaults on most platforms to *0x80000000*. When a persistence file is reloaded, the same base address must be specified as when the persistence file was first created. A base address of zero may be specified. This allows the computer to allocate the actual address, which it then displays. This base address must then be used when the database is reloaded. This option should be used along with the -s option to specify a reasonable excess of free space.
-b <filename>	Overrides the default filename used for persistent storage with *filename*. The default filename is *ifr_default_backing_store*.
`-f`	Use flat file persistence as an alternative to the memory mapped file persistence (use this option instead of '-p'. For example, `IFR_Service -f -b my_flat_file.dat` If the *-f* option is used in the absence of the *-b* option, then the IFR_Service will create and store its data in a file called *ifr_default_backing_store*.
-m	Enables read-write locking of IFR calls. If the IFR is started up with multi-threading enabled, for example if a service configuration file is used that specifies thread-per-connection, then this option should be used. Note that if *ACE_HAS_THREADS* is not defined, then this option will be ignored.

Table 3 Interface Repository Command Line Options

Option	Description
`-o <filename>`	Overrides the default filename used for storing the Interface Repository IOR. The default filename is *if_repo.ior*.
`-p`	Makes the Interface Repository persistent.
`-r`	Uses the Win32 registry for the database. Not available with persistence. The *-r* option is ignored if the *-p* option is used. If the platform is not Win32, an error message is output.
`-s <initial_size>`	This option only works when you also specify the *-p* option. The `-s` option allocates the amount of free space for the memory mapped persistent database to grow in to. May be specified in decimal, octal (with a leading zero) or hexadecimal (with a leading 0x). Only affects the initial size of the persistent database files when this file is first created. When the free space already allocated is exhausted, and attempt is made to allocate more space. This attempt is not guaranteed to succeed.

2.2 Administration

2.2.1 tao_ifr

This is the executable that administers the IFR. Calling *tao_ifr <filename>* will add the contents of the IDL file to the repository.

Calling *tao_ifr -r <filename>* removes the contents of the IDL file from the repository.

`tao_ifr` requires all the libraries that are required by the IFR service, plus the `IFR_Service` executable itself.

`tao_ifr` can also handle the *-ORBxxx* parameters, where the *xxx* represents a particular *ORB* parameter, for example:

```
-ORBInitRefInterfaceRepository=file://<filename>
```

ORBInitRefInterfaceRepository enables the IFR service to be resolved by getting its IOR from *<filename>*.

By default, the IFR service stores its IOR in the file *if_repo.ior*, but that can be modified by starting the IFR service using the *-o* option (see above).

All *-ORBxxx* options appear in the command line before any other options.

tao_ifr can process multiple IDL files in one execution. As long as the file names come after any -ORB options that may be present, they may come mixed in any order with the other command line options. The tao_ifr command line parser will treat any option (or option pair) that doesn't begin with a hyphen (-) as a filename.

Table 4 Legal tao_ifr Command Line Options

Option	Description
-Cw	Warning if identifier spellings differ only in case (default is error).
-Ce	Error if identifier spellings differ only in case (default).
-d	Outputs (to stdout) a dump of the AST.
-Dname[=value]	Defines name for preprocessor.
-E	Runs preprocessor only, prints on stdout.
-Idir	Includes *dir* in search path for preprocessor.
-L	Enables locking at the IDL file level.
-r	Removes contents of IDL file(s) from repository.
-Si	Suppresses processing of included IDL files.
-t	Temporary directory to be used by the IDL compiler.
-Uname	Undefines name for preprocessor.
-A...	Local implementation-specific escape.
-u	Prints usage message and exits.
-v	Traces compilation stages.
-w	Suppresses IDL compiler warning messages.
-Yp, path	Defines the location of the preprocessor.

CHAPTER

3 Naming Service

3.1 Introduction

The Naming Service provides a straightforward way for application components to find and using objects by associating meaningful names with them. The Naming Service can then be used like a white pages telephone directory to find an object and obtain its Object Reference, without complex programming or using proprietary ORB mechanisms.

3.1.1 OMG Standard

The Naming Service associates meaningful names with objects. An association between a name and an object's Interoperable Object Reference (IOR) is called a *binding* or *name binding*.

Name bindings are grouped in hierarchies called *naming contexts*. A naming context is an object containing zero or more name bindings. Each name binding within a naming context refers to either another naming context or a CORBA object.

There is no limit to the number of different names that can be bound to the same object or naming context, or to the number of bindings that a naming context can contain.

Resolving a name is the process of locating an object or naming context by reading a name binding and retrieving the associated object reference.

Iteration is the process of retrieving a list of bindings from a naming context, and looking at each binding in turn.

3.1.1.1 Naming Contexts

A naming context is a set of name bindings where each name is unique within that context; the same name may, however, appear in other naming contexts. Naming contexts can be bound to other naming contexts to create naming hierarchies.

A very simple hierarchy of naming contexts is shown in *Figure 1*. It illustrates the fact that a given binding within a naming context can point to either an object or another naming context, and that a single object can be referenced by more than one name. These hierarchies are known as *naming graphs*.

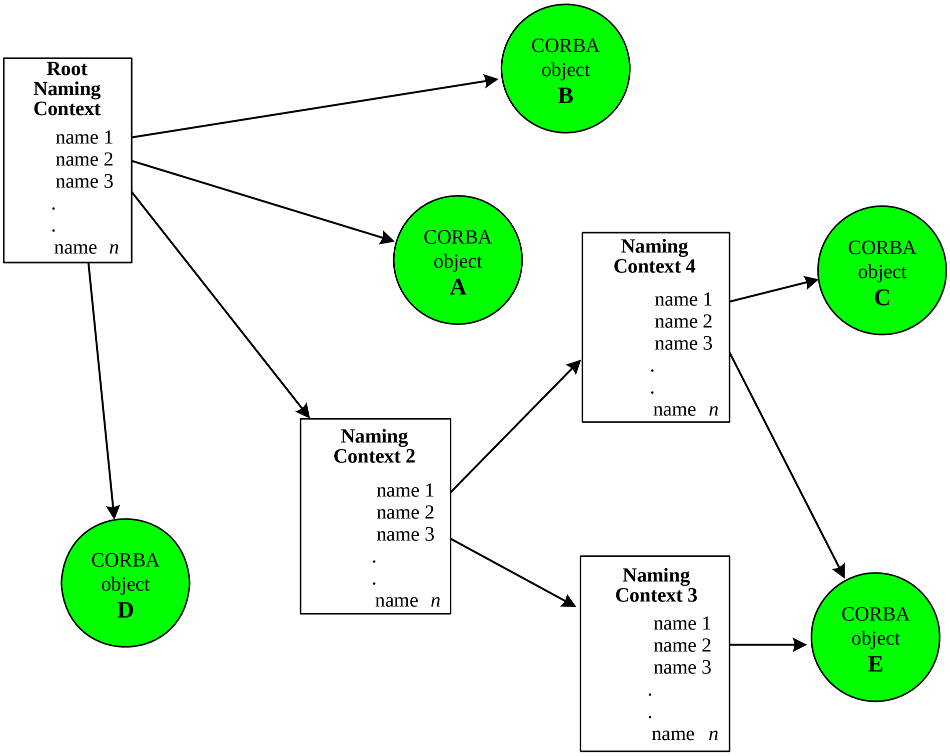


Figure 1 Simple Naming Graph

An object is referenced using an *initial* naming context, which is also referred to as the *root* context. This is followed by a sequence of one or more *name components*. Such a sequence is known as a *compound name*. Each name component resolves to the next naming context in a chain until the last name component resolves to the required object. In *Figure 1*, objects A, B and D are bound directly to the root context, so their names have only one component (these are *simple names*); objects C and E have names with three components. The full compound name for object C can be represented like this:

```
NamingContext2/NamingContext4/ObjectC
```

Object E can be accessed via two different names.

The service specification also permits a naming context to contain a binding which refers to a parent or grandparent further up the graph. For example, in *Figure 1* Naming Context 4 could contain a binding to Naming Context 2. This kind of reference is sometimes referred to as *cyclic*.

The root context is always implicit in a compound name; a special operation, `resolve_initial_references`, is performed once to obtain the root context, and all subsequent `resolve` operations depend on that.

Although it is not a requirement of the service specification, it is convenient and customary to have a single root naming context.

3.1.1.2 Name Components

Each name component has *id* and *kind* fields (sometimes referred to as *attributes*), represented by IDL strings. These strings are composed of ISO Latin-1 characters (excluding the ASCII *NUL*, 00h) and the combined length can be up to 255 characters.

The Naming Service always matches names using both fields, so it is acceptable for either field to be zero-length or to contain an empty string provided that uniqueness within a naming context is maintained. *Table 5* shows valid combinations of id and kind values.

Table 5 Name Component Fields

Id	Kind
name1	<empty>
name2	kind1
<empty>	<empty>
<empty>	kind2

Note that although it is technically possible for both fields to contain empty strings, this is not normally recommended, as it can be confusing to resolve to an empty name.

3.1.2 TAO Naming Service and Persistence

The TAO Naming Service instance can be run in either a *non-persistent* or *persistent* mode. The service runs in non-persistent mode by default.

When the service is run in the non-persistent mode, the data which the instance uses to associate names to objects is only retained in volatile memory: if the service instance stops for any reason, then this data is lost. Applications which are using the service instance’s name bindings for object resolution will then be unable to locate the objects, with the expected unfortunate consequences.

When the service is run in persistent mode, the data which the service uses to associate names to objects is saved to persistent storage. The service then is able to re-establish the instance’s name bindings after the service been has stopped and restarted. Applications which are using the service instance’s name bindings for object resolution will continue to be able to locate objects using the existing name bindings.

3.2 Running the Service

The Naming Service can be started by running the *`Naming_Service`* program from the command line. The `Naming_Service`’s command line options are listed in Table 6, *Naming_Service Command Line Options*. The `Naming_Service` program is located in the *`bin`* directory of the OpenFusion TAO distribution.

WIN The Naming Service can also be run as an Windows NT service (see Section 3.4, *Running as a Windows NT Service*, on page 27).

Table 6 Naming_Service Command Line Options

Option	Description
-b <base_address>	This option should only be used with the *-f* option. The *-b* option specifies a non-default virtual memory address which is used to map the shared memory mapped file into memory and This enables multiple name servers to be run on the same machine, each using different shared memory mapped file base addresses. This option should be used consistently for any given persistent store since it is not possible to relocate an existing persistent store. The base addresses should be sufficiently far apart in order to provide adequate memory for size of mapped files which might be use. The -u, -r or -v options are greatly preferred persistence mode alternatives which should be considered instead of the -f and -b option combination due to its complexity of usage and being error prone.
-d	Enables printing debug information to stdout. This option is the equivalent using *-ORBDebugLevel 1*. The debug level is increased for each instance that -d is given on the command line. For example, using "*-d -d -d*" will raise the debug level to *3*.
-f <persistent_file>	Specifies that the server will run in persistence mode using a shared memory map stored in the file *<persistent_file>.* The server's data and state is persisted to *<persistent_file>*. This option is ***not*** recommended. The *-u*, *-r* and *-v* options are the preferred persistence mode alternatives. This option can be used with the *-b* option to override the default memory address which the persistence file is mapped to. Note that the service's internal memory structures are copied directly to the file and subsequently are dependent on the platform, software and build versions, and other critical factors.

Table 6 Naming_Service Command Line Options (Continued)

Option	Description
-m <0|1>	The *-m* option enables or disables the Naming Service to respond to multicast requests. TAO uses a simple, non-standard method for clients to discover the Naming Service's initial reference. This can be inadequate and cause unexpected results if, for example, there are multiple naming services running on the network. The Naming Service's default behaviour is not to respond to multicast queries (use the Interoperable Naming Service bootstrap options instead). The -m option enables the service to respond to multicast requests. *1* = enable multicast responses, *0* = disable (default).
-ORBDebugLevel <level>	Enables printing debug information to stdout. The debug level is set to the value of *<level>*. Using *-ORBDebugLevel 1* is the equivalent of using the *-d* option.
-ORBEndPoint <endpoint>	Specifies where the Naming Service server will listen for requests from clients, where *<endpoint>* is location specified as *iiop://tcp_hostname:port*.
-ORBNameServicePort <nsport>	Specifies or overrides the Naming Service's multicast network port it will listen to for multicast requests. This option is only used when multicast option is enabled with *-m 1*.
-o <ior_file>	Enables the root context's IOR to be stored in the file *<ior_file>*. Clients can use the IOR stored in the file to locate the service's root context.
-p <pid_file>	Enables the service's server process id (PID) to be stored in the file *<pid_file>*. The stored PID can be used (on UNIX systems) for stopping or *killing* the naming service server daemon.
-s <context_size>	This option is used to set the size of the hash table allocated for the root context when a new root context is created. All contexts created under the root will use the same size for their hash tables. This option can be used for basic performance tuning with the binding name look-up if the number of anticipated root entries is known. This option does not limit the size of the persistent store but will effect its size when the store is created. The default value is 1024.

Table 6 Naming_Service Command Line Options (Continued)

Option	Description
`-r <directory_path>`	Specifies that the server should run in persistence mode using flat file persistence. The server's data will be stored in files placed in the directory <directory_path>. This option allows multiple running name services to share the same persistent data files in order to achieve simple service redundancy and replication. Using this option incurs additional file locking overhead: if only one name service instance is to be using the data set then the -u option should be preferred. This option is an alternative to using the `-f`, `-u` or `-v` persistence options.
`-t <time>`	Specifies that the service will terminate if a client request has not been received by it within `<time>` seconds. If a request is reviewed with the time interval, then the service will continue to run. The default behaviour if for the service to run indefinitely (i.e. when `-t` is not used).
`-u <directory_path>`	Specifies that the server should run in persistence mode using flat directory persistence. The server's data will be stored in files placed in the directory <directory_path>. This option is an alternative to using the `-f`, `-r` or `-v` persistence options.
`-v <directory>`	Specifies that the server should run in persistence mode using flat file persistence. Each entity in the naming graph is repesented by an individual file stored in a directory tree, where `<directory>` is the tree's root directory.
`-z <time>`	Sets the round trip timeout value (in seconds) for operations using a federated naming context. If the time interval is exceeded the "Cannot proceed" exception is thrown to the client. The default behaviour if for the service to not timeout (i.e. when `-z` is not used). ⚠ This option relies on the use of file system soft links and is accordingly not available on win32-based systems such as Windoes NT or XP).

Example

This example starts the Naming Service, enables multicasting using port 1122 for listening, and saves its IOR to a file called name.ior

```
%  Naming_Service -m 1 -ORBNameServicePort 1122 -o name.ior
```

3.2.1 Environment Variables

The `NameServicePort` environment variable is set to the multicast port used by clients who want to bootstrap to a Naming Service using multicast. This environment variable is used only when multicast responding is enabled (using the command line option `-m 1`).

3.2.2 Persistence Options

The Naming Service's persistence mode version is set using one, and only one, of the persistence mode command-line options, *-f*, *-r*, *-u*, or *-v*, when starting the server. Each option persists Naming Service's data and state to one or more files, where.

- if the file(s) do not exist, the they are created
- if the file(s) exist, then the Naming Service's state is set to the state stored in the file(s).

The behaviour of some of these options can be controlled using other command line options, including using the

- the *-b* with the *-f* option to set the based virtual memory address
- and the *-s* option to set hash table size for any of the persistent mode alternatives

3.2.2.1 The *-f* Option

The *-f* option maps the service's internal memory, which contains the service's current state, to a file. A default memory address (`ACE_DEFAULT_BASE_ADDR`) is used for mapping the file. Alternate mapping address can be specified at compile-time by redefining `TAO_NAMING_BASE_ADDR` in `tao/orbconf.h`. Alternate mapping address can also be specified at run-time with the `-b` command-line option, which takes precedence over `TAO_NAMING_BASE_ADDR` definition.

⚠ The Naming Service stores absolute pointers in its memory-mapped file. Therefore, it is important to use the same mapping address on each run for the same persistence file.

3.2.3 Implementation Policies

3.2.3.1 Destroying Binding Iterators

A binding iterator is destroyed when client invokes the *destroy* operation either on the iterator itself or on the naming context it is iterating over. In both cases, subsequent calls on the binding iterator object will cause *OBJECT_NOT_EXIST* exception.

3.2.3.2 Orphaned Naming Contexts

This implementation of the Naming Service does not remove or free resources consumed by orphaned naming contexts when the service is running in an persistent mode: it is the client's responsibility to clean up and remove naming contexts and in order to avoid leaking server resources.

However, when the service is running in non-persistent mode then resources, including orphaned contexts, are released when the Naming Server is shutdown.

3.2.4 Bootstrapping the Naming Service from Clients

There are several methods which a client can use to connect to a Naming Service server instance, in other words, there are several mechanisms which *resolve_initial_references* can use when asked for "*NameService*". In order of predictable behaviour, they are:

1. Using command-line options

   The *-ORBInitRef NameService=IOR:...* or environment variable *NameServiceIOR* can be used on the client side to specify the object that the call to should return to the client. (On the server side, *-o* option can be used to get the IOR).

Example (UNIX, same host):

```
%   TAO_ROOT/orbsvcs/Naming_Service -o ior_file
%   my_client -ORBInitRef NameService=file://ior_file
```

   On the first line, we start the Naming Service, and output its IOR to *ior_file*. On the second line, we start some client, and specify the IOR *resolve_initial_references* should return for the Naming Service in a file format.

2. Using Multicast

   When started with the *respond to multicast queries option* turned on (*-m 1*), clients can use IP multicast to query for a naming service, and this instance will respond. The Naming Server is listening for client multicast requests on a specified port. On the client side, sends out a multicast request on the network, trying to locate a Naming Service. When a Naming Server receives a multicast request from a client, it replies to the sender with the IOR of its root naming context.

The port used for this bootstrapping process, i.e., the multicast port, has nothing to do with the ORB port used for CORBA communication. Other points worth mentioning include:

- A client and a server can communicate using the multicast protocol if they are using the same multicast port. For both client and server *-ORBnameserviceport* command-line option and *NameServicePort* environment variable can be used to specify the multicast port to use. If none is specified, the default port is used. (The ability to specify multicast ports can be used to match certain clients with certain Naming Servers, when there are more than one Naming Server running on the network).
- If there are several naming servers running on the network, each listening on the same port for multicast requests, each will send a reply to a client's request. The client's orb will use the first response it receives, so the Naming Service will, in fact, be selected at random.

Since this mechanism is proprietary to TAO (i.e., non-standard), it only works when both client and server are written using TAO. There is no way to turn multicasting off on the client side, but it is used only as a last resort, i.e., any of the other options will override it.

3.3 Administration

The following command line utilities can be used administer objects which are, or need to be, registered with the Naming.

3.3.0.1 nslist

nslist displays a formatted list of current Naming Service entries and is run from the command line as follows:

```
%  nslist [ --ior | --nsior ]
```

where

nslist displays the contents of the default "*NameService*" (including the protocol and end point of each object reference) returned by resolve_initial_references()

--ior is an optional switch which displays the contents of the NameService, including the IOR of each reference entry and the IOR of the NameService itself

`--nsior` is an optional switch which displays ***only*** the IOR of the NameService itself, with no other text. This switch can be used to locate the TAO NameService for non-TAO applications.

3.3.0.2 nsadd

`nsadd` can be used to add objects into the Name Service and is run as follows:

```
%  nsadd < --name obj_name > < --ior ior > [ --rebind ]
```

where

`--name obj_name` adds an object identified by `obj_name` (required)

`--ior ior` supplies the object's IOR contained in the file identified by `ior` (required)

`--rebind` optionally rebinds the object (as per the Naming Service's `rebind` method)

3.3.0.3 nsdel

`nsdel` deletes objects from the Name Service. It is run from the command line using:

```
%  nsdel < --name obj_name >
```

where

`--name obj_name` specifies the object identified by `obj_name` to be removed (required)

3.4 Running as a Windows NT Service

WIN The Naming Service can be run as a Windows NT service by running `NT_Naming_Service.exe` from the DOS command line. `NT_Naming_Service` must be run using the command line options described in *Table 7*.

⚠ The command line options listed in *Table 6* can not be used as a command line option with `NT_Naming_Service`. The options listed in *Table 6* are specified using Windows Registry Keys, described in *Step 2:*.

The `NT_Naming_Service.exe` utility is provided with the OpenFusion TAO Naming Service distribution.

Step 1: Install the service to NT by running `NT_Naming_Service` with the `-i` option:

```
>  NT_Naming_Service -i
```

Step 2: Add the Naming Service's start-up options to Windows Registry.

The start-up options that are specified for the standard Naming_Service command line must be set in Windows Registry before the NT_Naming_Service is started. These command line options take affect whenever NT_Naming_Service is run, unless the settings are deleted or modified. Create the registry key described below to set the Naming Service's command line options. The Windows Registry is edited using Windows' Registry Editor, *`regedit`*. `regedit` can be run by entering *`regedit`* into Windows' **Start | Run** dialog or by running it directly from a Windows command line prompt.

Add the following registry key using the Registry Editor:

> Path: *`My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\ACE\TAO`*
>
> Key: *`TaoNamingServiceOptions`*
>
> Type: *`REG_SZ`* (Zero terminated String)
>
> Value: *`-ORBEndPoint iiop://213.48.91.6:10005 -o c:\temp\name.ior -f c:\temp\naming_service.dat`* (Note: this is an example. You should provide the actual values specific to your installation)

If this key is not in the Registry, then NT_Naming_Service will run as if no command line options were given.

Step 3: Start the service by running `NT_Naming_Service` with the *`-s`* option. Note that the service must be started in order to operate.

```
>  NT_Naming_Service -s
```

To stop the service run `NT_Naming_Service` with the *`-k`* option. The service can be restarted by re-running `NT_Naming_Service` with the *`-s`* option.

```
>  NT_Naming_Service -k
```

All of the available `NT_Naming_Service` command line options are listed in *Table 7*. The service can be controlled using the Windows NT Services Control Panel after installation (this is the recommended method).

Table 7 NT_Naming_Service Command Line Options

`-i`	Install the Naming Service as a Windows NT Service.
`-s`	Starts the service using the standard Naming Service options as specified in the Windows Registry. See *Step 2:* on page 27.
`-t [value]`	Sets the service's Windows NT start-up type behaviour (**Manual**, **Automatic** or **Disabled**). Values are: *Automatic* - 2 *Manual* - 3 *Disabled* - 4
`-r`	Remove the service. Ensure the service has been stopped before removing.
`-k`	Stop (or *kill*) the service (the service remains installed).

CHAPTER

4 Event Service

4.1 Introduction

The OMG Event Service is a service for decoupling the suppliers of events from the consumers. For many applications, this approach provides a much more appropriate model than the synchronous invocation mechanism of CORBA.

4.2 Running the Service

The Event Service is started with the *CosEvent_Service* command, optionally followed by any of the options listed in Table 8, *CosEvent_Service Command-line Options*.

Table 8 CosEvent_Service Command-line Options

Option	Description	Default
`-n COS_EC_name`	Specifies the name with which to bind the event channel (in the root naming context of the Naming Service). Ignored if the `-x` option is used.	`CosEventService`
`-r`	Use the `rebind()` operation to bind the event channel in the Naming Service. If the name is already bound and this flag is not passed, the process exits with an `Already Bound` exception. Ignored if the `-x` option is used.	The `bind()` operation is used.
`-x`	Do not use the Naming Service. This simply creates an event channel.	Bind the event channel in the Naming Service.

The *CosEvent_Service* server supplies the capability to start a single event channel in its own process. It can bind the created event channel to a supplied name in the root naming context of the Naming Service. The Naming Service must be running before the *CosEvent_Service* server is started, unless the `-x` command-line option is used. The created event channel implements the `CosEventChannelAdmin::EventChannel` interface.

When the `destroy()` operation of the event channel is called, the process exits and the event channel is unbound from the naming service.

4.3 Event Channel Configuration

4.3.1 Run-time Configuration

The new implementation of the COS Event Service uses a factory to build all the objects and strategies it requires. The factory can be dynamically loaded using ACE Service Configurator. This is extremely convenient because the factory can also parse options in the Service Configurator script file.

The current implementation provides a default implementation for this factory This document describes the options used by this default implementation. Users can define their own implementation with new ad-hoc strategies or with pre-selected strategies.

4.3.2 The Configuration File

The COS channel uses the same service configurator file that the ORB uses. The default name for this file is *`svc.conf`* but the ORB option `-ORBSvcConf` can be used to override this. The format of the file is described in detail in the service configurator documentation but the relevant section for the event channel looks like this:

```
# Comments go here...
# More comments if you want to...
static CEC_Factory "-CECDispatching reactive ....."
```

All the event service factory options start with `-CEC`

4.3.3 Options

Table 9 Event Channel Configuration Options

Option	Description
`-CECDispatching` *`dispatching_strategy`*	Select the dispatching strategy used by the COS event service. A *reactive* strategy will use the same thread that received the event from the supplier to push the event to all the consumers. The *mt* strategy will also use a pool of threads, but the thread to dispatch is randomly selected.
`-CECDispatchingThreads` *`number_of_threads`*	Select the number of threads used by the *mt* dispatching strategy.
`-CECProxyConsumerLock` *`lock_type`*	Select the lock type (*null*, *thread* or *recursive*) to synchronize access to the ProxyPushConsumer state.

Table 9 Event Channel Configuration Options (Continued)

Option	Description
`-CECProxySupplierLock` *`lock_type`*	Select the lock type (*null*, *thread* or *recursive*) to synchronize access to the ProxyPushSupplier state.
`-CECUseORBId` *`orbid`*	Set the name of the ORB used by the event service. Only useful in applications that create multiple ORBs and activate the event service in one of them.
`-CECConsumerControl` *`policy`*	Select the consumer control policy (*null* or *reactive*) to detect and discard broken consumers.
`-CECSupplierControl` *`policy`*	Select the supplier control policy (*null* or *reactive*) to detect and discard broken suppliers.
`-CECConsumerControlPeriod` *`period`*	Set the period (in microseconds) used by the reactive consumer control policy to poll the state of the consumers.
`-CECSupplierControlPeriod` *`period`*	Set the period (in microseconds) used by the reactive supplier control policy to poll the state of the suppliers.
`-CECConsumerControlTimeout` *`timeout`*	Set the timeout period (in microseconds) used by the reactive consumer control policy to detect a timeout when polling the state of the consumers.
`-CECSupplierControlTimeout` *`timeout`*	Set the timeout period (in microseconds) used by the reactive supplier control policy to detect a timeout when polling the state of the suppliers.
`-CECReactivePullingPeriod` *`period`*	Set the period (in microseconds) used by the reactive pulling strategy to poll all the PullSuppliers for an event.
`-CECProxyConsumerCollection` *`flag[:flags]`*	Configure the data structure and strategies used to implement collections of `ProxyPushConsumers` and `ProxyPullConsumers`. The argument is a colon separated list of flags, described in Table 10, *Proxy Collection Flags*.
`-CECProxySupplierCollection` *`flag[:flags]`*	Configure the data structure and strategies used to implement collections of `ProxyPushSupplier` and `ProxyPullSupplier` objects. The argument is a colon separated list of flags, described in Table 10, *Proxy Collection Flags*.

Table 10 Proxy Collection Flags

Flag	Description
`MT`	Use regular mutexes and/or condition variables for serialization.
`ST`	Use null mutexes and/or condition variables for serialization.
`LIST`	Implement the collection using an ordered list, fast for iteration (i.e. during event dispatching), but slow for insertion and removal (i.e. when clients connect and disconnect from the EC).
`RB_TREE`	Implement the collection using a Red-Black tree, slow for iteration (i.e. during event dispatching), but fast for insertion and removal (i.e. when clients connect and disconnect from the EC).
`IMMEDIATE`	Threads block until they can execute a change on the data structure, the system must use other approaches to guarantee that the iterators are not invalidated during event dispatching. For example, use a separate dispatching thread. Using this option with the reactive values for any of the `-CECSupplierControl`, `-CECConsumerControl`, or `-CECDispatching` options may cause deadlocks.
`COPY_ON_READ`	Before initiating an iteration to dispatch events (or similar tasks) a copy of the complete collection is performed. This solves most of the synchronization problems, but introduces a significant source of overhead and priority inversions on the critical path.
`COPY_ON_WRITE`	Similar to the previous one, but the copy is only performed when needed.
`DELAYED`	Threads that need to change the collection can detect if that change will invalidate iterators used by other threads. If so, the thread posts the change on a queue that is executed once the collection is no longer in use.

4.3.4 The Constructor

The TAO_CEC_EventChannel class implements the CosEventChannelAdmin::EventChannel interface. This class takes one mandatory and two optional parameters in its constructor:

```
TAO_CEC_EventChannel (const TAO_CEC_EventChannel_Attributes&
attributes,
                    TAO_CEC_Factory* factory = 0,
                    int own_factory = 0);
```

The factory is an optional parameter to override the default strategy factory used by the event channel. The event channel will destroy the factory if the own_factory argument is true.

The attributes parameter can be used to fine tune some of the algorithms and strategies used by the event channel. The default values are probably OK for most applications. Notice that the attributes include the POA used to activate the ConsumerAdmin, SupplierAdmin, ProxyPushConsumer, ProxyPushSupplier, ProxyPullConsumer and the ProxyPullSupplier objects. These POAs must have the IMPLICIT_ACTIVATION and the SYSTEM_ID policies (as the RootPOA does). See Table 11, *Constructor Attributes* for a list of allowed attributes.

Table 11 Constructor Attributes

Attribute	Description
consumer_reconnect	If the attribute is not zero then the same consumer can call connect_push_consumer on its ProxyPushSupplier multiple times to change its subscriptions. This is usually more efficient that disconnecting and connecting again.
supplier_reconnect	If the attribute is not zero then the same supplier can call connect_push_supplier on its ProxyPushConsumer multiple times to change its publications. This is usually more efficient that disconnecting and connecting again.
disconnect_callbacks	It not zero the event channel will send disconnect callbacks when a disconnect method is called on a Proxy. In other words, if a consumer calls disconnect_push_supplier() on its proxy the event channel will invoke disconnect_push_consumer() on the consumer. A similar thing is done for suppliers.
busy_hwm	When the delayed flag is set on proxy collections, this flag controls the maximum number of threads that can simultaneously iterate over the collection before blocking. It can be used to avoid starvation in delayed updates on the collection.

Table 11 Constructor Attributes

Attribute	Description
`max_write_delay`	When the delayed flag is set on proxy collections, this flag controls the maximum number of threads that will initiate dispatching after a change has been posted. Any thread after that is blocked until the operations are performed. It can be used to completely stop starvation of delayed updates on the collection.
`supplier_poa`	The POA used by the event channel to activate SupplierAdmin and SupplierProxy objects.
`consumer_poa`	The POA used by the event channel to activate ConsumerAdmin and ConsumerProxy objects.

CHAPTER

5 Utilities

5.1 Descriptions and Usage

The following utilities can be used to perform tasks which can help users manage and use TAO more easily.

5.1.1 catior

`catior` reads the IOR stored in file (a stringified IOR), decodes it and sends the contents to `stdout` using:

```
%  catior -f filename
```

where

`-f filename` reads the file identified by `filename` (containing the IOR).

Example

This example shows `catior` being used to display the IOR which is stored in the `NotificationSingleton.ior` file of the current directory.

```
%  catior -f NotificationSingleton.ior
```

The output IOR is shown below.

```
IOR:000000000000004B49444C3A707269736D742E636F6D2F636F732F436F734E6F7469666
9636174696F6E2F4E6F74696669636174696F6E457874656E73696F6E732F51756575654D61
6E616765723A312E300000000000002000000000000000B4000102000000000E3231332E34382
E39312E3230360004C70000005F4F70656E467573696F6E2E4E6F74696669636174696F6E53
6572766963652F4F70656E467573696F6E2E4E6F74696669636174696F6E536572766963652
F218D4798E0DE1811D7A75ABC571A2C16BF8A426F30DE1811D7A75ABC571A2C16BF00000000
020000000000000008000000004A414300000000010000001C000000000501000100000001 0
501000100010109000000010501000100000001000000 2C00000000000000010000000100 00
001C00000000050100010000000105010001000101090000000105010001

decoding an IOR:

The Byte Order: Big Endian

The Type Id:
"IDL:prismt.com/cos/CosNotification/NotificationExtensions/Queue
```

```
Manager:1.0"

Number of Profiles in IOR:      2

Profile number: 1

IIOP Version:   1.2

    Host Name:  213.48.91.206

    Port Number:        1223

    Object Key len:     95

    Object Key as hex:

    4f 70 65 6e 46 75 73 69 6f 6e 2e 4e 6f 74 69 66  69 63 61 74 69 6f 6e 53
65 72 76 69 63 65 2f 4f    70 65 6e 46 75 73 69 6f 6e 2e 4e 6f 74 69 66 69 63
61 74 69 6f 6e 53 65 72 76 69 63 65 2f 21 8d    47 98 e0 de 18 11 d7 a7 5a bc
57 1a 2c 16 bf 8a 42 6f 30 de 18 11 d7 a7 5a bc 57 1a 2c 16 bf

    The Object Key as string:

OpenFusion.NotificationService/OpenFusion.NotificationService/!.G.......Z.W
.,...Bo0.....Z.W.,..

    The component <1> ID is 0 (TAG_ORB_TYPE)

          ORB Type: 1245790976

    The component <2> ID is 1 (TAG_CODE_SETS)

          Component Value len:  28

          Component Value as hex:

          00 00 00 00 05 01 00 01 00 00 00 01 05 01 00 01 00 01 01 09 00 00
00 01 05 01 00 01

          The Component Value as string:

          ............................

Profile number: 2

    Profile tag = 1 (unknown protocol)

    Profile body len:   44

    Profile body as hex:
       00 00 00 00 00 00 00 01 00 00 00 01 00 00 00 1c 00 00 00 00 05 01 00 01
   00 00 00 01 05 01 00 01 00 01 01 09 00 00 00 01 05 01 00 01

       The Profile body as string:

       ............................................

%  catior returned true
```

5.1.2 ior-parser

The *ior-parser* utility parses IORs generated by most ORBs. It has been tested with Orbix, VisiBroker and TAO. `ior-parser` is used as follows:

```
%  ior-parser <IOR filename>
```

5.1.3 gperf

gperf is a GNU perfect hash function generator which is used by TAO's IDL compiler to generate perfect hash functions (which is generally the most efficient and predictable operation demuxing technique).

A complete description of `gperf` and how to use it, along with its GNU Licensing Terms & Conditions, is provided in *gperf.pdf* which is located in the *docs/release/pdf* directory of the OpenFusion TAO distribution.

Index

Index