User Manual

COM2CORBA

Date: 2009-05-08
Version: 1.0.0 (revision 424)

This document

Summary This document presents the user manual for the COM2CORBA project.

Author Marcus Ericsson

Date 2009-05-08

Version 1.0.0 (revision 424)

Document history

Version	Date	Changes
0.1.0	2009-05-07	Document created
1.0.0	2009-05-08	First version

Contents

Chapter 1

User Manual Tao Options

1.1 Document overview

The document is a copy of TAO IDL Compiler User's Guide[1].

Appendix A

A.1 Scope

This document describes the options and features of TAO IDL compiler. It is not a reference manual or tutorial on OMG IDL. For more information on OMG IDL see the online CORBA specification[2] and the Advanced CORBA Programming with C++[3] book. More information on the design of TAO's IDL compiler is also available online[4]. Finally, comprehensive documentation on TAO's IDL compiler is available in the OCI TAO Developer's Guide[5].

A.2 Generated Files

The IDL compiler generates 9 files from each .idl file. The file names are obtained by taking the IDL basename and appending the following suffixes (see the list of TAO's IDL compiler options[6] on how to get different suffixes for these files:)

- ***Client stubs***, *i.e.*, *C.h, *C.inl, and *C.cpp. Pure client applications only need to #include and link with these files.
- ***Server skeletons***, *i.e.*, *S.h, *S.inl, and *S.cpp. Servers need to #include and link with these files.
- ***Server skeleton templates***, *i.e.*, *S_T.h, S_T.inl, and *S_T.cpp. Some C++ compilers do not like template and non-template code in the same files, so TAO's IDL compiler generates these files separately.

TAO's IDL compiler creates separate *.inl and *S_T.* files to improve the performance of the generated code. For example, the *.inl files enable you to compile with inlining enabled or not, which is useful for trading-off compile-time and run-time performance. Fortunately, you only need to #include the client stubs declared in the *C.h file and the skeletons in the *S.h file in your code.

A.3 Environment Variables

Variable	Usage
TAO_IDL_PREPROCESSOR	Used to override the program name of the preprocessor that 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 TAO_IDL doesn't have any code to implement a preprocessor, it has to 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"

And the same behavior 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.). And case 3 isn't used at all, 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 overriden 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 one of these. It is possible that support for these old variables will be removed in a future version of TAO.

If ACE_ROOT or TAO_ROOT are defined, then TAO_IDL will use them to include the $(ACE_ROOT)/TAO/tao or $(TAO_ROOT)/tao directories. 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.

A.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[7]. To generate perfect hash functions, TAO's IDL compiler uses gperf[8] , which is a general-purpose perfect hash function generator.

To configure TAO's IDL compiler to support perfect hashing please do the following:

- Enable ACE_HAS_GPERF when building ACE and TAO. This macro has been defined for the platforms where gperf has been tested, which includes most platforms that ACE runs on

- Build the gperf in $ACE_ROOT/apps/gperf. This build also leaves a copy/link of the gperf program at the $ACE_ROOT/bin directory.
- Set the environment variable $ACE_ROOT appropriately or add $ACE_ROOT/bin to your search path.
- Use the -g option for the TAO IDL compiler or set your search path accordingly to install gperf in a directory other than $ACE_ROOT/bin.

Note that if you can't use perfect hashing for some reason the next best operation demuxing strategy is binary search, which can be configured using TAO's IDL compiler options[9].

A.5 AMI support

The TAO_IDL compiler generates AMI stubs and skeletons as described in the CORBA 3.0.3 specification.

A.6 Collocation Strategies

TAO_IDL can generate collocated stubs using two different collocation strategies. It also allows you to supress/enable the generation of the stubs of a particular strategy. To gain great flexibility at run-time, you can gereate stubs for both collocation strategies (using both '-Gp'[10] and '-Gd'[11] flags at the same time) and defer the determination of collocation strategy until run-time. On the other hand, if you want to minimize the footprint of your program, you might want to pre-determine the collocation strategy you want and only generate the right collocated stubs (or not generating any at all using both '-Sp'[12] and '-Sd'[13] flags at the same time if it's a pure client.) See our collocation paper[14] for a detail discussion on the collocation support in TAO.

A.7 TAO's IDL Compiler Options

TAO's IDL compiler invokes your C (or C++) preprocessor to resolve included IDL files. It receives the common options for preprocessors (such as -D or -I). It also receives other options that are specific to it.

Option	Description	Remark
-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 backend.	
	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, which 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.
	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, which 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.
	anyop_export_macro=macro_name	The compiler will emit macro_name before each Any operator or extern typecode declaration in the generated stub code, this is needed for Windows, which requires special directives to export symbols from DLLs, usually the definition is just a space on unix platforms. This option works only in conjuction with the -GA option, which generates Any operators and typecodes into a separate set of files.

Option	Description	Remark
	anyop_export_include=include_path	The compiler will generate code to include include_path at the top of the anyop file header, this is usually a good place to define the export macro. This option works in conjuction with the -GA option, which generates Any operators and typecodes into a separate set of files.
	export_macro=macro_name	This option has the same effect as issuing
		-Wb,skel_export_macro =macro_name
		-Wb,stub_export_macro= macro_name
		-Wb,anyop_export_macro= macro_name.
		This option is useful when building a DLL containing both stubs and skeletons.
	export_include=include_path	This option has the same effect as specifying
		-Wb,stub_export_include= include_path
		-Wb,skel_export_include= include_path
		-Wb,anyop_export_include= include_path.
		This option goes with the previous option to build DLL containing both stubs and skeletons.

Option	Description	Remark
	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 precompiled header mechanism, such as those provided by Borland C++Builder or MSVC++.
	obv_opt_accessor	The IDL compiler will generate code to optimize access to base class data for valuetypes.
	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 pushes 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 pops 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.
	include_guard=define	The compiler will generate code the define in the C.h file to prevent users from including the generated C.h file. Useful for regenerating the pidl files in the archive.
	safe_include=file	File that the user should include instead of this generated C.h file. Useful for regenerating the pidl files in the archive.
	unique_include=file	File that the user should include instead of the normal includes in the C.h file. Useful for regenerating the *_include pidl files in the archive.

Option	Description	Remark
-E	Invoke only the preprocessor	
-Wp, option_list	Pass options to the preprocessor.	
-d	Causes output of a dump of the AST	
-Dmacro _definition	It is passed to the preprocessor	
-Umacro_name	It is passed to the preprocessor	
-Iinclude_path	It is passed to the preprocessor	
-Aassertion	It is passed to the preprocessor	
-Yp,path	Specifies the path for the C preprocessor	
-H perfect_hash	To specify the IDL compiler to generate skeleton code that uses perfect hashed operation demuxing strategy, which is the default strategy. Perfect hashing uses gperf program, to generate demuxing methods.	

Option	Description	Remark
-H dynamic_hash	To specify the IDL compiler to generate skeleton code that uses dynamic hashed operation demuxing strategy.	
-H binary_search	To specify the IDL compiler to generate skeleton code that uses binary search based operation demuxing strategy.	
-H linear_search	To specify the IDL compiler to 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 hasing program (GPERF). Default is $ACE_ROOT/bin/gperf.	
-o path	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.	If the specified directory does not exist, it will be created, if any path that may precede the directory name already exists. If the directory itself already exists, no action is taken.
-oS path	Same as -o option but applies only to generated *S.* files	Default is value of -o option or current directory
-oA path	Same as -o option but applies only to generated *A.* files	Default is value of -o option or current directory

Option	Description	Remark
-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.inl".	
-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.inl".	
-t	Temporary directory to be used by the IDL compiler.	Unix: use environment variable TEMPDIR if defined, else use /tmp/. Windows NT/2000/XP: use environment variable TMP or TEMP 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 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.
-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)	
-GC	Generate AMI stubs ("sendc_" methods, reply handler stubs, etc)	
-GH	Generate AMH stubs, skeletons, exception holders, etc.	
-Gp	Generated collocated stubs that use Thru_POA collocation strategy (default)	
-Gd	Generated collocated stubs that use Direct collocation strategy	
-Gce	Generated code for CORBA/e. This reduces the size of the generated code	
-Gmc	Generated code for Minimum CORBA. This reduces the size of the generated code	
-Gsp	Generate client smart proxies	

Option	Description	Remark
-Gt	Generate optimized TypeCodes	
-GX	Generate empty A.h file	Used by TAO developers for generating an empty A.h file when the -GA option can't be used. Overrides -Sa and -St.
-Guc	Generate uninlined constant if defined in a module	Inlined (assigned a value in the C++ header file) by default, but this causes a problem with some compilers when using pre-compiled headers. Constants declared at global scope are always generated inline, while those declared in an interface or a valuetype never are - neither case is affected by this option.
-Gse	Generate explicit export of sequence's template base class	Occasionally needed as a workaround for a bug in Visual Studio (.NET 2002, .NET 2003 and Express 2005) where the template instantiation used for the base class isn't automatically exported
-GI	Generate boiler-plate files that contain empty servant implementations	
-GIh arg	Servant implementation header file name ending	
-GIs arg	Servant implementation skeleton file name ending	
-GIb arg	Prefix to the implementation class names	
-GIe arg	Suffix to the implementation class names	

Option	Description	Remark
-GIc	Generate copy constructors in the servant implementation template files	
-GIa	Generate assignment operators in the servant implementation template files	
-GId	Generate IDL compiler source file/line# debug info in implementation files	
-GT	Enable generation of the TIE classes, and the *S_T.* files that contain them.	
-GA	Generate type codes and Any operators in *A.h and *A.cpp	Decouples client and server decisions to compile and link TypeCode- and Any-related code, which is generated in *C.h and *C.cpp by default. If -Sa or -St also appear, then an empty *A.h file is generated.
-Gos	Generate std::ostream insertion operators for IDL declarations	ORB IDL declarations (including the basic type sequences) don't have these operators generated by default, to avoid the increased footprint. To turn on this generation for ORB IDL files, set gen_ostream=1 in your default.features MPC file. If this option is used on application IDL that references any of the predefined basic sequence IDL types, TAO must be compiled with the gen_ostream feature turned on.
-Sa	Suppress generation of the Any operators	
-Sal	Suppress generation of the Any operators for local interfaces only	
-Sp	Suppress generation of collocated stubs that use Thru_POA collocation strategy	

Option	Description	Remark
-Sorb	Suppress generation of the ORB.h include.	This option is useful when regenerating pidl files in the core TAO libs to prevent cyclic includes;
-Se	Disable custom header file name endings for files that are found in TAO specific include directories (i.e. $TAO_ROOT, $TAO_ROOT/tao, $TAO_ROOT/orbsvcs, $TAO_ROOT/CIAO, $TAO_ROOT/CIAO/ciao, $TAO_ROOT/CIAO/ccm).	include directories (i.e. $TAO_ROOT, $TAO_ROOT/tao, $TAO_ROOT/orbsvcs, $TAO_ROOT/CIAO, $TAO_ROOT/CIAO/ciao, $TAO_ROOT/CIAO/ccm). This option is useful when used together with -hs or -hc. I.e. when user needs custom file name endings for his/her own files but still wants to use TAO specific files with their original endings;

Bibliography

[1] "http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html" http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html

[2] http://www.omg.org/technology/documents/corba_spec_catalog.htmhttp://www.omg.org/technology/documents/corba_spec_catalog.htm

[3] http://www.triodia.com/staff/michi-henning.htmlhttp://www.triodia.com/staff/michi-henning.html

[4] http://www.cs.wustl.edu/~schmidt/PDF/ami1.pdfhttp://www.cs.wustl.edu/~schmidt/PDF/ami1.pdf

[5] http://www.theaceorb.com/product/index.htmlhttp://www.theaceorb.com/product/index.html

[6] http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#optionshttp://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#options

[7] http://www.cs.wustl.edu/~schmidt/PDF/COOTS-99.pdfhttp://www.cs.wustl.edu/~schmidt/PDF/COOTS-99.pdf

[8] http://www.cs.wustl.edu/~schmidt/PDF/gperf.pdfhttp://www.cs.wustl.edu/~schmidt/PDF/gperf.pdf

[9] http://www.cs.wustl.edu/~schmidt/ACE-versions-i.htmlhttp://www.cs.wustl.edu/~schmidt/ACE-versions-i.html

[10] http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Gphttp://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Gp

[11] http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Gdhttp://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Gd

[12] http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Sphttp://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Sp

[13] http://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Sdhttp://www.dre.vanderbilt.edu/~schmidt/DOC_ROOT/TAO/docs/compiler.html#Sd

[14] http://www.cs.wustl.edu/~schmidt/PDF/C++-report-col18.pdfhttp://www.cs.wustl.edu/~schmidt/PDF/C++-report-col18.pdf