IDL PreProcessor Guide

Release 6.x

CONTENTS

1 Preface **1**
- 1.1 About the IDL PreProcessor Guide . . . 1
- 1.2 Intended Audience . . . 1
- 1.3 Organisation . . . 1
- 1.4 Conventions . . . 2

2 Overview **3**
- 2.1 Introduction . . . 3

3 Prerequisites **5**

4 Command Line Options **6**

5 OpenSplice Modes and Languages **10**

6 IDL Pre-processor Grammar **11**

7 Keys **16**
- 7.1 Defining Keys . . . 16
  - 7.1.1 Supported types for keys . . . 16
  - 7.1.2 Character arrays as Keys . . . 17
- 7.2 Bounded strings as character arrays . . . 18

8 Modes, Languages and Processing steps **19**
- 8.1 Integrated C++ ORB . . . 19
- 8.2 C++ Standalone . . . 21
- 8.3 ISOC++ . . . 21
- 8.4 ISOC++2 . . . 21
- 8.5 C Standalone . . . 21
- 8.6 C99 Standalone . . . 23
- 8.7 Java Standalone . . . 23
- 8.8 Integrated Java ORB . . . 25

9 Extensible and Dynamic Topic Types for DDS annotation support **26**

10 Built-in DDS data types **28**

11 References **29**

12 Contacts & Notices **31**
- 12.1 Contacts . . . 31
- 12.2 Notices . . . 32

CHAPTER

ONE

PREFACE

1.1 About the IDL PreProcessor Guide

The *IDL Pre-processor Guide* describes what the Vortex OpenSplice IDL Pre-processor is, and how to use it.

The Vortex OpenSplice IDL Pre-processor is included with the Vortex OpenSplice product.

1.2 Intended Audience

The *IDL Pre-processor Guide* is intended to be used by developers creating applications which use Vortex OpenSplice.

1.3 Organisation

The *Overview* gives a general description of and brief introduction to the IDL Pre-processor.

Prerequisites describes the prerequisites needed to run the pre-processor.

IDL Pre-processor Command Line Options gives detailed descriptions of the options that are available for running the pre-processor.

OpenSplice Modes and Languages provides a summary of OpenSplice's supported modes and languages, as well as an overview of the applicable Vortex OpenSplice libraries.

IDL Pre-processor Grammar shows the IDL grammar that is supported by the Vortex OpenSplice IDL Pre-processor.

Keys describes the mechanism for the use of keys with particular data types.

Modes, Languages and Processing steps describes the steps required for creating programs for each of the modes and languages supported by the Pre-processor.

Extensible and Dynamic Topic Types for DDS annotation support describes how the IDL Pre-processor handles the annotation language extension.

Built-in DDS data types describes the built-in DDS data types and provides language-specific guidelines on how to use them.

Finally, there is *a bibliography* which lists all of the publications referred to in this *Guide*.

1.4 Conventions

The icons shown below are used in ADLINK product documentation to help readers to quickly identify information relevant to their specific use of Vortex OpenSplice.

Icon	*Meaning*
	Item of special significance or where caution needs to be taken.
	Item contains helpful hint or special information.
	Information applies to Windows (*e.g.* XP, 2003, Windows 7) only.
	Information applies to Unix-based systems (*e.g.* Solaris) only.
	Information applies to Linux-based systems (*e.g.* Ubuntu) only.
	C language specific.
	C++ language specific.
	C# language specific.
	Java language specific.

CHAPTER

TWO

OVERVIEW

The Vortex OpenSplice IDL Pre-processor plays a role in generating code for DDS/DCPS specialized interfaces (TypeSupport, DataReader and DataWriter) from application data definitions defined in IDL for all supported languages.

2.1 Introduction

The Vortex OpenSplice IDL Pre-processor supports two modes:

- **Standalone** mode where the application is only used with Vortex OpenSplice
- **ORB-integrated** mode where the application is used with an ORB as well as with Vortex OpenSplice

In a *standalone* context, Vortex OpenSplice provides, apart from the DDS/DCPS related artifacts, all of the artifacts implied by the lDL language-specific mapping. In this case the name space used is `DDS` instead of the name space implied by the IDL language-specific mapping.

In an *ORB-integrated* context, the ORB pre-processor will provide for the artifacts implied by the lDL language-specific mapping, while Vortex OpenSplice only provides the DDS/DCPS-related artifacts. The application data type representation provided by the ORB is also used within the Vortex OpenSplice context. In this way application data types can be shared between the ORB and Vortex OpenSplice within one application program.

The Vortex OpenSplice IDL Pre-processor accepts IDL which complies with the OMG CORBA specification to specify application data types. Additionally it allows specifying keys on data types.

A number of DDS data types defined in the DCPS API (for example, `Time_t`) are available for use with application IDL data types and can be seen as OpenSplice DDS IDL Pre-processor "built-in" definitions.

The diagram *OpenSplice IDL Pre-processor High Level Processes* shows the Vortex OpenSplice IDL Pre-processor high-level processing.

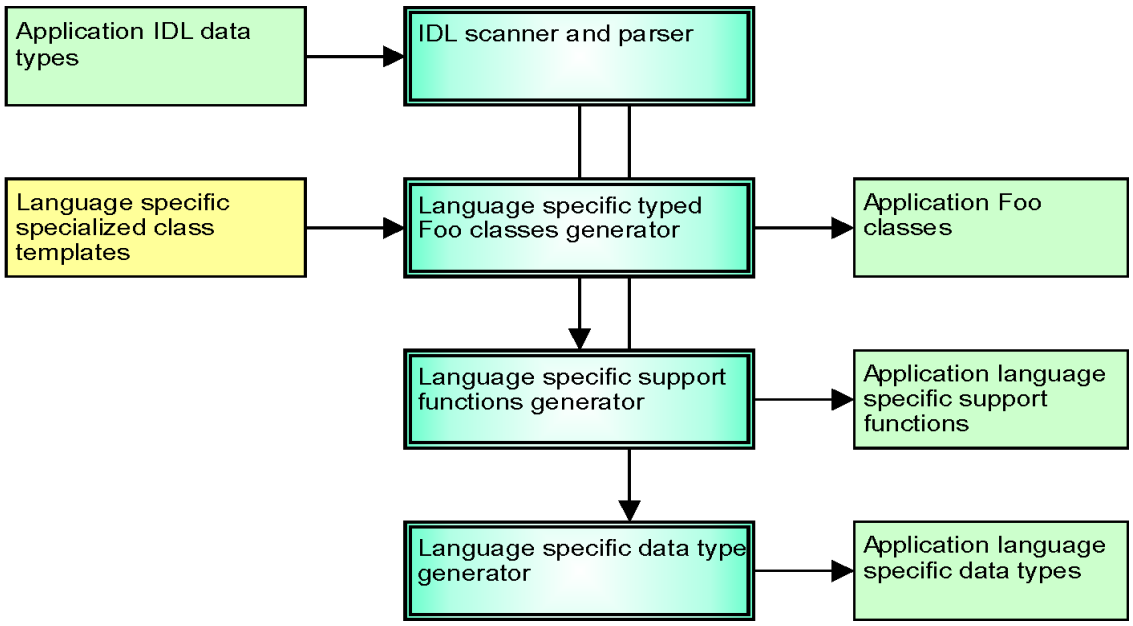
The Vortex OpenSplice IDL Pre-processor scans and parses the IDL input file containing the application data type definitions.

For the selected language, the Vortex OpenSplice IDL Pre-processor generates the specialized interfaces for *TypeSupport*, the *DataReader* and the *DataWriter* from specialized class template files which are provided by OpenSplice. Note that the Vortex OpenSplice IDL Pre-processor will only generate specialized interfaces for application data types for which a key list is defined. If it is not, the OpenSplice DDS IDL Pre-processor assumes that the data type will only be used enclosed in other data types.

The Vortex OpenSplice IDL Pre-processor also generates language-specific support functions, which are needed to allow the Vortex OpenSplice system to handle the application data types.

For the *standalone* context the Vortex OpenSplice IDL Pre-processor generates the language-specific application data types according the OMG IDL language mapping that is applicable for the specific target language.

OpenSplice IDL Pre-processor High Level Processes


Application IDL data types
IDL scanner and parser
Language specific specialized class templates
Language specific typed Foo classes generator
Application Foo classes
Language specific support functions generator
Application language specific support functions
Language specific data type generator
Application language specific data types

CHAPTER

THREE

PREREQUISITES

Unix

The Vortex OpenSplice environment must be set correctly for UNIX-based platforms before the Vortex OpenSplice IDL Pre-processor can be used.

Run `release.com` from a shell command line to set the environment.

`release.com` is located in the root directory of the Vortex OpenSplice installation (<OSPL_HOME>):

```
% . <OSPL_HOME>/release.com
```

The Vortex OpenSplice IDL Pre-processor, `idlpp`, can be invoked by running it from a command shell:

```
% idlpp
```

The `idlpp` command line options are fully described in *IDL Pre-processor Command Line Options*.

CHAPTER

FOUR

COMMAND LINE OPTIONS

The Vortex OpenSplice IDL Pre-processor, `idlpp`, can be run with the following command line options:

```
[ -h ]
[ -b <ORB-template-path> ]
[ -n <include-suffix> ]
[ -I <path> ]
[ -D <macro>[=<definition>] ]
< -S | -C >
< -l (c | c++ | cpp | java | cs | isocpp | isoc++ | c99 | simulink) >
[ -F ]
[ -j [old]:<new>]
[ -o <dds-types> | <custom-psm> | <no-equality> | <deprecated-c++11-mapping>
      | <disable-file-header>]
[ -d <output-directory> ]
[ -P <dll_macro_name>[,<header_file>] ]
[ -N ]
<filename>
```

- Options shown between angle brackets, < and >, are mandatory.
- Options shown between square brackets, [and], are optional.

All of these options are described in full detail below.

-h List the command line options and information.

-b <ORB-template-path> Specifies the ORB-specific path within the template path for the specialized class templates (in case the template files are ORB specific). The ORB-specific template path can also be set *via* the environment variable `OSPL_ORB_PATH`, the command line option is however leading. To complete the path to the templates, the value of the environment variable `OSPL_TMPL_PATH` is prepended to the ORB path.

-n <include-suffix> Overrides the suffix that is used to identify the ORB-dependent header file (specifying the data model) that needs to be included. Normally the name of this include file is derived from the IDL file name and followed by an ORB-dependent suffix (*e.g.* `C.h` for ACE-TAO based ORBs). This option is only supported in CORBA cohabitation mode for C++; in all other cases it is simply ignored. Example usage: `-n .stub.hpp` (For a file named `foo.idl` this will include `foo.stub.hpp` instead of `fooC.h`, which is the default expectation for ACE-TAO.)

-I <path> Passes the include path directives to the C pre-processor.

-D <macro> Passes the specified macro definition to the C pre-processor.

-S Specifies standalone mode, which allows application programs to be built and run without involvement of any ORB. The name space for standard types will be DDS instead of the name space implied by the IDL language mapping. types will be DDS instead of the name space implied by the IDL language mapping.

-C Specifies ORB integrated mode, which allows application programs to be built and run integrated with an ORB.

-l (c | c++ | cpp | java | cs | isocpp | isoc++ | isocpp2 | isoc++2 | c99 | simulink) Selects the target language. Note that the Vortex OpenSplice IDL Pre-processor does not support every combination of modes and languages. This option is mandatory; when no language is selected the OpenSplice IDL Pre-processor reports an error.

- For the *c*, *c++*, *cpp*, *java* and *cs* target languages the types will default to the standard types. For the *isocpp* and *isoc++* target languages the types will default to the ISOC++ types that comply with the ISO/IEC C++ 2003 Language DDS PSM. When using *isocpp*, *isoc++* *isocpp2* or *isoc++2* an equality operator will also be generated for types unless this feature is explicitly disabled.
- Please note that *isocpp* and *isoc++* target languages are DEPRECATED since V6.6.0. Please use *isocpp2* or *isoc++2* instead.
- For the Standalone mode in C (when using the `-S` flag and the `c` language option), `OSPL_ORB_PATH` will by default be set to value SAC, which is the default location for the standalone C specialized class template files.
- For the CORBA cohabitation mode in C++ (when using the `-C` flag and the `c++` or `cpp` language option) the `OSPL_ORB_PATH` will, by default, be set to:

  `CCPP/DDS_OpenFusion_1_6_1` for Unix-based platforms.

  `CCPP\DDS_OpenFusion_1_6_1` for Windows platforms.

  These are the default locations for the IDL to C++ specialized class template files of the OpenSplice-Tao ORB. Class templates for other ORBS are also available in separate sub-directories of the CCPP directory, but for more information about using a different ORB, consult the `README` file in the `custom_lib/ccpp` directory.
- For the Standalone mode in C++ (when using the `-S` flag and the `c++` or `cpp` language option), `OSPL_ORB_PATH` will by default be set to value `SACPP`, which is the default location for the standalone C++ specialized class template files.

- For the Standalone mode in Java (when using the `-S` flag and the `java` language option), `OSPL_ORB_PATH` will by default be set to the value of `SAJ`, which is the default location for the standalone Java specialized class template files.
- For the CORBA cohabitation mode in Java (when using the `-C` flag and the `java` language option), `OSPL_ORB_PATH` will by default be set to the value of `SAJ`, which is the default location for the CORBA Java specialized class template files. This means that the CORBA cohabitated Java API and StandAlone Java API share the same template files.

- For the Standalone mode in C# (when using the `-S` flag and the `cs` language option), `OSPL_ORB_PATH` will by default be set to the value of `SACS`, which is the default location for the standalone CSharp specialized class template files.

  |c99|
- For the c99 target language the types will default to the standard types. Except that the primitive types are mapped to the corresponding c99 types and that bound strings are mapped to char arrays with a size one larger than specified in the idl definition to allow for the terminating 0 character.

  |simulink|

- For Simulink a MATLAB .m file is created representing the simulink bus for the input IDL file. Typically this option is used when invoking a script from MATLAB to import the IDL into Simulink.

See also *OpenSplice Modes and Languages* for a complete list of supported modes and languages.

-F Specifies FACE mode, generate FACE API type specific functions in addition to the target language specific ones. *Only applicable for the java and isocpp2 target languages.*

Java

-j [old]:<new> Specifies that the (partial) package name which matches *[old]* will be replaced by the package name which matches *<new>* (the package *<new>* is substituted for the package *[old]*). If *[old]* is not included then the package name defined by *<new>* is prefixed to all Java packages. The package names may only be separated by `.` (period) characters. A trailing `.` character is not required, but may be used. Example: `-j :org.opensplice` (prefixes all Java packages). Example: `-j com.opensplice.:org.opensplice.` (substitutes). *Only applicable for the Java language.*

-o dds-types Enables the built-in DDS data types. In the default mode, the built-in DDS data types are not available to the application IDL definitions. When this option is activated, the built-in DDS data types will become available. Refer to Section 1.9, Built-in DDS data types, on page 28.

-o custom-psm Enables support for alternative IDL language mappings. Currently CSharp offers an alternative language mapping where IDL names are translated to their PascalCase representation and where `@` instead of `_` is used to escape reserved C#-keywords.

-o no-equality Disables support for the automatically-generated equality operator on ISOC++ types.

-o deprecated-c++11-mapping Generates the ISOC++2 types using the deprecated C++11 mapping implementation as used in the past by the also deprecated isocpp/isoc++ PSM. This option only makes sense when migrating from isocpp/isoc++ to isocpp2/isoc++2.

-o disable-file-header Disables adding the common 'fileHeaderContents' template to the start of each generated output file. By default, when this option is *not* provided, the contents of the template file 'fileHeaderContents' located in the directory 'Common' in `OSPL_TMPL_PATH` will be added to the start of each generated file. This template can be modified to customize the header contents. The following variables can be used in this template and will be replaced by the actual values:

- **$(idl_filename)** The file name of the IDL input file that is used for generating output
- **$(opensplice_version)** The OpenSplice version
- **$(timestamp)** Current date and time

Note that the template should not contain any comment delimiters (e.g. /* .. */). For each output file the language specific comment delimiters are added automatically.

-d <output-directory> Specifies the output directory for the generated code.

-P <dll_macro_name>[,<header_file>] This option controls the signature for every external function/class interface. If you want to use the generated code for creating a DLL, then interfaces that need to be accessible from the outside need to be exported. When accessing these operations outside of the DLL, then these external interfaces need to be imported. If the generated code is statically linked, this option can be omitted. The first argument *<dll_macro_name>* specifies the text that is prepended to the signature of every external function and/or class. For example: defining DDS_API as the macro, the user can define this macro as `__declspec(dllexport)` when building the DLL containing the generated code, and define the macro as `__declspec(dllimport)` when using the DLL containing the generated code.

Addtionally a header file can be specified, which contains controls to define the macro. For example the external interface of the generated code is exported when the macro `BUILD_MY_DLL` is defined, then this file could look like:

```
#ifdef BUILD_MY_DLL
#define DDS_API __declspec(dllexport)
#else /* !BUILD_MY_DLL */
#define DDS_API __declspec(dllimport)
#endif /* BUILD_MY_DLL */
```

C C++

-N This option disables type caching in the copy-in routines. The copy-in routines cache the type to improve the performance of copying sequences. This option disables this feature to allow the use of sequences within multi-domain applications. *Only applicable for the C and C++ languages.*

<filename> Specifies the IDL input file to process.

CHAPTER

FIVE

OPENSPLICE MODES AND LANGUAGES

The Vortex OpenSplice IDL Pre-processor supports two modes:

- *Standalone* mode where the application is only used with Vortex OpenSplice
- *ORB-integrated* mode where the application is used with an ORB as well as with Vortex OpenSplice

In a *standalone* context, Vortex OpenSplice provides, apart from the DDS/DCPS related artifacts, all the artifacts implied by the lDL language specific mapping. In this case the used name space is DDS instead of the name space implied by the IDL language specific mapping.

In an *ORB-integrated* context, the ORB pre-processor will provide for the artifacts implied by the lDL language specific mapping, while Vortex OpenSplice only provides the DDS/DCPS related artifacts. The application data type representation provided by the ORB is also used within the Vortex OpenSplice context. In this way application data types can be shared between the ORB and Vortex OpenSplice within one application program.

The languages and modes that Vortex OpenSplice supports are listed in the table below.

Supported Modes and Languages

Language	Mode	OpenSplice Library	ORB Template Path
C	Standalone	dcpssac.so dcpsac.lib	SAC
C++	ORB Integrated	dcpsccpp.so	CCPP/DDS_OpenFusion_1_4_1 *for UNIX-like platforms, and* CCPP\DDS_OpenFusion_1_5_1 *for the Windows platform*
C++	Standalone	dcpssacpp.so	SACPP
ISOC++	ISOCPP Types	dcpsisocpp.so	ISOCPP
ISOC++	ORB Integrated	dcpsisocpp.so	CCPP/DDS_OpenFusion_1_4_1 *for UNIX-like platforms, and* CCPP\DDS_OpenFusion_1_5_1 *for the Windows platform*
ISOC++2	ISOCPP Types	dcpsisocpp2.so	ISOCPP2
Java	Standalone	dcpssaj.jar	SAJ
Java	ORB integrated	dcpscj.jar	SAJ
C#	Standalone	dcpssacs Assembly. dll	SACS
C99	Standalone	dcpsc99.so dcpsc99.lib	C99

The mappings for each language are in accordance with their respective OMG Language Mapping Specifications (see *the Bibliography* for a list of references).

CHAPTER

SIX

IDL PRE-PROCESSOR GRAMMAR

The Vortex OpenSplice IDL Pre-processor accepts the grammar which complies with the CORBA Specification. The Vortex OpenSplice IDL Pre-processor accepts the complete grammar, but it will ignore elements not relevant to the definition of data types.

In the following specification of the grammar (similar to EBNF), elements that are processed by the Vortex OpenSplice IDL Pre-processor are highlighted in **bold**. Note that Vortex OpenSplice does not support all base types that are specified by the OMG.

The `idlpp` also takes into account all C pre-processor directives that are common to ANSI-C, like `#include`, `#define`, `#ifdef`, *etc.*.

The OMG's IDL Grammar

<specification>	::=	<import>* **<definition>+**
<definition>	::=	<type_dcl> ";" <ann_appl_post> | **<type_dcl>** ";" | **<const_dcl>** ";" | <except_dcl> ";" | <interface> ";" | **<module>** ";" | <value> ";" | <type_id_dcl> ";" | <type_prefix_dcl> ";" | <event> ";" | <component> ";" | <home_dcl> ";" | <annotation> ";" <ann_appl_post>
<annotation>	::=	<ann_dcl> | <ann_fwd_dcl>
<ann_dcl>	::=	<ann_header> "{" <ann_body> "}"
<ann_fwd_dcl>	::=	"@Annotation ["(" ")"] local interface" <identifier>
<ann_header>	::=	"@Annotation ["(" ")"] local interface" <identifier>
<ann_body>	::=	<ann_attr>*
<ann_inheritance_spec>	::=	":" <annotation_name>
<annotation_name>	::=	<scoped_name>
<ann_attr>	::=	<ann_appl> "attribute" <param_type_spec> <simple_declarator> ["default" <const_exp>] ";" <ann_appl_post>
<ann_appl>	::=	{ "@" <ann_appl_dcl> }*
<ann_appl_post>	::=	{ "//@" <ann_appl_dcl> }*
<ann_appl_dcl>	::=	<annotation_name> ["(" [<ann_appl_params>] ")"]
<ann_appl_params>	::=	<const_exp> | <ann_appl_param> { "," <ann_appl_param> }*
<ann_appl_param>	::=	<identifier> "=" <const_exp>
<struct_header>	::=	<ann_appl> "struct" <identifier> [":" <scoped_name>]
<switch_type	::=	<integer_type> | <char_type> | <wide_char_type> | <boolean_type> | <enum_type> | <octet_type> | <scoped_name>
<map_type>	::=	"map" "<" <simple_type_spec> "," <ann_appl> <simple_type_spec> "," <ann_appl_post> <positive_int_const> ">" | "map" "<" <simple_type_spec> "," <ann_appl> <simple_type_spec> <ann_appl_post> ">"
<module>	::=	**"module" <identifier> "{" <definition>+ "}"**
<interface>	::=	<interface_dcl> | <forward_dcl>

continues on next page

Table 1 – continued from previous page

<interface_dcl>	::=	<interface_header> "{" <interface_body> "}"
<forward_dcl>	::=	["abstract" | "local"] "interface" <identifier>
<interface_header>	::=	["abstract" | "local"] "interface" <identifier> [<interface_inheritance_spec>]
<interface_body>	::=	<export>*
<export>	::=	<type_dcl> ";" | <const_dcl> ";" | <except_dcl> ";" | <attr_dcl> ";" | <op_dcl> ";" | <type_id_dcl> ";" | <type_prefix_dcl> ";"
<interface_inheritance_spec>	::=	":" <interface_name> { "," <interface_name> }*
<interface_name>	::=	<scoped_name>
<scoped_name>	::=	**<identifier> | "::" <identifier> | <scoped_name> "::" <identifier>**
<value>	::=	(<value_dcl> | <value_abs_dcl> | <value_box_dcl> | <value_forward_dcl>)
<value_forward_dcl>	::=	["abstract"] "valuetype" <identifier>
<value_box_dcl>	::=	"valuetype" <identifier> <type_spec>
<value_abs_dcl>	::=	"abstract" "valuetype" <identifier> [<value_inheritance_spec>] "{" <export>* "}"
<value_dcl>	::=	<value_header> "{" < value_element>* "}"
<value_header>	::=	["custom"] "valuetype" <identifier> [<value_inheritance_spec>]
<value_inheritance_spec>	::=	[":" ["truncatable"] <value_name> { "," <value_name> }*] ["supports" <interface_name> { "," <interface_name> }*]
<value_name>	::=	<scoped_name>
<value_element>	::=	<export> | < state_member> | <init_dcl>
<state_member>	::=	("public" | "private")<type_spec> <declarators>";"
<init_dcl>	::=	"factory" <identifier> "(" [<init_param_decls>] ")" [<raises_expr>] ";"
<init_param_decls>	::=	<init_param_decl> { "," <init_param_decl> }*
<init_param_decl>	::=	<init_param_attribute> <param_type_spec> <simple_declarator>
<init_param_attribute>	::=	"in"
<const_dcl>	::=	**"const" <const_type> <identifier> "=" <const_exp>**
<const_type>	::=	**<integer_type>** | **<char_type>** | <wide_char_type> | **<boolean_type>** | **<floating_pt_type>** | **<string_type>** | **<wide_string_type>** | **<fixed_pt_const_type>** | ****<scoped_name>** | **<octet_type>**
<const_exp>	::=	**<or_expr>**
<or_expr>	::=	**<xor_expr> | <or_expr> "|" <xor_expr>**
<xor_expr>	::=	**<and_expr> | <xor_expr> "^" <and_expr>**
<and_expr>	::=	**<shift_expr> | <and_expr> "&" <shift_expr>**
<shift_expr>	::=	**<add_expr> | <shift_expr> ">>" <add_expr> | <shift_expr> "<<" <add_expr>**
<add_expr>	::=	**<mult_expr> | <add_expr> "+" <mult_expr> | <add_expr> "-" <mult_expr>**
<mult_expr>	::=	**<unary_expr> | <mult_expr> "*" <unary_expr> | <mult_expr> "/" <unary_expr> | <mult_expr> "%" <unary_expr>**
<unary_expr>	::=	**<unary_operator> <primary_expr> | <primary_expr>**
<unary_operator>	::=	**"-" | "+" | "~"**
<primary_expr>	::=	**<scoped_name> | <literal> | "(" <const_exp> ")"**
<literal>	::=	**<integer_literal>** | **<string_literal>** | <wide_string_literal> | **<character_literal>** | <wide_character_literal> | <fixed_pt_literal> | **<floating_pt_literal>** | <boolean_literal>
<boolean_literal>	::=	**"TRUE" | "FALSE"**

continues on next page

Table 1 – continued from previous page

<positive_int_const>	::=	**<const_exp>**
<type_dcl>	::=	**"typedef" <type_declarator>** | **<struct_type>** | **<union_type>** | **<enum_type>** | "native" <simple_declarator> | <constr_forward_decl>
<type_declarator>	::=	**<type_spec> <declarators>**
<type_spec>	::=	**<simple_type_spec>** | **<constr_type_spec>**
<simple_type_spec>	::=	**<base_type_spec>** | **<template_type_spec>** | **<scoped_name>**
<base_type_spec>	::=	**<floating_pt_type>** | **<integer_type>** | **<char_type>** | <wide_char_type> | **<boolean_type>** | **<octet_type>** | <any_type> | <object_type> | <value_base_type>
<template_type_spec>	::=	**<sequence_type>** | **<string_type>** | <wide_string_type> | <fixed_pt_type> | <map_type>
<constr_type_spec>	::=	**<struct_type>** | **<union_type>** | **<enum_type>**
<declarators>	::=	**<declarator> { "," <declarator> }***
<declarator>	::=	**<simple_declarator>** | **<complex_declarator>**
<simple_declarator>	::=	**<identifier>**
<complex_declarator>	::=	**<array_declarator>**
<floating_pt_type>	::=	**"float"** | **"double"** | "long" "double"
<integer_type>	::=	**<signed_int>** | **<unsigned_int>**
<signed_int>	::=	**<signed_short_int>** | **<signed_long_int>** | **<signed_longlong_int>**
<signed_short_int>	::=	**"short"**
<signed_long_int>	::=	**"long"**
<signed_longlong_int>	::=	**"long" "long"**
<unsigned_int>	::=	**<unsigned_short_int>** | **<unsigned_long_int>** | ****<unsigned_longlong_int>**
<unsigned_short_int>	::=	**"unsigned" "short"**
<unsigned_long_int>	::=	**"unsigned" "long"**
<unsigned_longlong_int>	::=	**"unsigned" "long" "long"**
<char_type>	::=	**"char"**
<wide_char_type>	::=	"wchar"
<boolean_type>	::=	**"boolean"**
<octet_type>	::=	**"octet"**
<any_type>	::=	"any"
<object_type>	::=	"Object"
<struct_type>	::=	**<struct_header> "{" <member_list> "}"**
<member_list>	::=	**<member>+**
<member>	::=	**<type_spec> <declarators> ";"** | **<ann_appl> <type_spec> <declarator> ";" <ann_appl_post>**
<union_type>	::=	**<ann_appl> "union" <identifier> "switch" "(" <switch_type_spec> ")" "{" <switch_body> "}"**
<switch_type_spec>	::=	**<ann_appl> <switch_type_name> <ann_appl_post>**
<switch_body>	::=	**<case>+**
<case>	::=	**<case_label>+ <element_spec> ";" <ann_appl_post**
<case_label>	::=	**"case" <const_exp> ":"** | **"default" ":"**
<element_spec>	::=	**<ann_appl> <type_spec> <declarator>**
<enum_type>	::=	**<ann_appl> "enum" <identifier> "{" <enumerator> { "," <ann_appl_post> <enumerator> }* <ann_appl_post> "}"**
<enumerator>	::=	**<ann_appl> <identifier>**

continues on next page

Table 1 – continued from previous page

<sequence_type>	::=	**“sequence” “<” <ann_appl> <simple_type_spec> “,” <ann_appl_post> <positive_int_const> “>” | “sequence” “<” <ann_appl> <simple_type_spec> <ann_appl_post> “>”**
<string_type>	::=	**“string” “<” <positive_int_const> “>” | “string”**
<wide_string_type>	::=	“wstring” “<” <positive_int_const> “>” | “wstring”
<array_declarator>	::=	**<identifier> <ann_appl> <ann_appl_post> <fixed_array_size>+**
<fixed_array_size>	::=	**“[” <positive_int_const> “]”**
<attr_dcl>	::=	<readonly_attr_spec> | <attr_spec>
<except_dcl>	::=	“exception” <identifier> “{” <member>* “}”
<op_dcl>	::=	[<op_attribute>] <op_type_spec> <identifier> <parameter_dcls> [<raises_expr>] [<context_expr>]
<op_attribute>	::=	“oneway”
<op_type_spec>	::=	<param_type_spec> | “void”
<parameter_dcls>	::=	“(” <param_dcl> { “,” <param_dcl> }* “)” | “(” “)”
<param_dcl>	::=	<param_attribute> <param_type_spec> <simple_declarator>
<param_attribute>	::=	“in” | “out” | “inout”
<raises_expr>	::=	“raises” “(” <scoped_name> { “,” <scoped_name> }* “)”
<context_expr>	::=	“context” “(” <string_literal> { “,” <string_literal> }* “)”
<param_type_spec>	::=	<base_type_spec> | <string_type> | <wide_string_type> | <scoped_name>
<fixed_pt_type>	::=	“fixed” “<” <positive_int_const> “,” <positive_int_const> “>”
<fixed_pt_const_type>	::=	“fixed”
<value_base_type>	::=	“ValueBase”
<constr_forward_decl>	::=	“struct” <identifier> | “union” <identifier>
<import>	::=	“import” <imported_scope> “;”
<imported_scope>	::=	<scoped_name> | <string_literal>
<type_id_dcl>	::=	“typeid” <scoped_name> <string_literal>
<type_prefix_dcl>	::=	“typeprefix” <scoped_name> <string_literal>
<readonly_attr_spec>	::=	“readonly” “attribute” <param_type_spec> <readonly_attr_declarator>
<readonly_attr_declarator>	::=	<simple_declarator> <raises_expr> | <simple_declarator> { “,” <simple_declarator> }*
<attr_spec>	::=	“attribute” <param_type_spec> <attr_declarator>
<attr_declarator>	::=	<simple_declarator> <attr_raises_expr> | <simple_declarator> { “,” <simple_declarator> }*
<attr_raises_expr>	::=	<get_excep_expr> [<set_excep_expr>] | <set_excep_expr>
<get_excep_expr>	::=	“getraises” <exception_list>
<set_excep_expr>	::=	“setraises” <exception_list>
<exception_list>	::=	“(” <scoped_name> { “,” <scoped_name> } * “)”
<component>	::=	<component_dcl> | <component_forward_dcl>
<component_forward_dcl>	::=	“component” <identifier>
<component_dcl>	::=	<component_header> “{” <component_body> “}”
<component_header>	::=	“component” <identifier> [<component_inheritance_spec>] [<supported_interface_spec>]
<supported_interface_spec>	::=	“supports” <scoped_name> { “,” <scoped_name> }*
<component_inheritance_spec>	::=	“:” <scoped_name>
<component_body>	::=	<component_export>*
<component_export>	::=	<provides_dcl> “;” | <uses_dcl> “;” | <emits_dcl> “;” | <publishes_dcl> “;” | <consumes_dcl> “;” | <attr_dcl> “;”
<provides_dcl>	::=	“provides” <interface_type> <identifier>
<interface_type>	::=	<scoped_name> | “Object”

continues on next page

Table 1 – continued from previous page

<uses_dcl>	::=	“uses” [“multiple”] < interface_type> <identifier>
<emits_dcl>	::=	“emits” <scoped_name> <identifier>
<publishes_dcl>	::=	“publishes” <scoped_name> <identifier>
<consumes_dcl>	::=	“consumes” <scoped_name> <identifier>
<home_dcl>	::=	<home_header> <home_body>
<home_header>	::=	“home” <identifier> [<home_inheritance_spec>] [<supported_interface_spec>] “manages” <scoped_name> [<primary_key_spec>]
<home_inheritance_spec>	::=	“:” <scoped_name>
<primary_key_spec>	::=	“primarykey” <scoped_name>
<home_body>	::=	“{” <home_export>* “}”
<home_export>	::=	<export> | <factory_dcl> “;” | <finder_dcl> “;”
<factory_dcl>	::=	“factory” <identifier> “(” [<init_param_decls>] “)” [<raises_expr>]
<finder_dcl>	::=	“finder” <identifier> “(” [<init_param_decls>] “)” [<raises_expr>]
<event>	::=	(<event_dcl> | <event_abs_dcl> | <event_forward_dcl>)
<event_forward_dcl>	::=	[“abstract”] “eventtype” <identifier>
<event_abs_dcl>	::=	“abstract” “eventtype” <identifier> [<value_inheritance_spec>] “{” <export>* “}”
<event_dcl>	::=	<event_header> “{” <value_element>* “}”
<event_header>	::=	[“custom”] “eventtype” <identifier> [<value_inheritance_spec>]
<identifier>	::=	Arbitrarily long sequence of ASCII alphabetic, numeric and underscore characters. The first character must be ASCII alphabetic. All characters are significant. An identifier may be escaped with a prepended underscore character to prevent collisions with new IDL keywords. The underscore does not appear in the generated output.

CHAPTER

SEVEN

KEYS

7.1 Defining Keys

The Vortex OpenSplice IDL Pre-processor also provides a mechanism to define a list of keys (space- or comma-separated) with a specific data type. The syntax for that definition is:

```
#pragma keylist <data-type-name> <key>*
```

The identifier <data-type-name> is the identification of a struct or a union definition.

The identifier <key> is the member of a struct. For a struct either no key list is defined, in which case no specialized interfaces (TypeSupport, DataReader and DataWriter) are generated for the struct, or a key list with or without keys is defined, in which case the specialized interfaces are generated for the struct. For a union either no key list is defined, in which case no specialized interfaces are generated for the union, or a key list without keys is defined, in which case the specialized interfaces are generated for the union. It is not possible to define keys for a union because a union case may only be addressed when the discriminant is set accordingly, nor is it possible to address the discriminant of a union. The keylist must be defined in the same name scope or module as the referred struct or union.

7.1.1 Supported types for keys

Vortex OpenSplice supports following types as keys:

- short
- long
- long long
- unsigned short
- unsigned long
- unsigned long long
- float
- double
- char
- boolean
- octet
- string
- bounded string

- enum
- char array (provided that `#pragma cats` is specified; see *Character arrays as Keys* below)

Vortex OpenSplice also supports typedef for these types.

7.1.2 Character arrays as Keys

By default Vortex OpenSplice does not support using a character array as a key. Using an (character) array as a key field is not desirable or supported, because:

1. Every index in the array must be considered a separate key in this situation. This is the only way that arrays can be compared to each other in a correct manner. An array of ten characters would have to be treated as a ten-dimensional storage structure, leading to poor performance compared with the processing of a (bounded) string of ten characters.
2. An array always has a fixed length and therefore the whole array is sent over the wire even if only a small part of it is needed. When using a (bounded) string, only the actual string is sent and not the maximum length.

However, in certain scenarios a character array is the logical key for a topic, either from an information modeling perspective or simply due to a legacy data model. To facilitate such scenarios Vortex OpenSplice introduces the following pragma which allows for character arrays to be used as a key.

```
#pragma cats <data-type-name> <char-array-field-name>*
```

The identifier `<data-type-name>` is the identification of a struct definition. The identifier `<char-array-field-name>` is the member of a struct with the type character array. The `cats` pragma *must* be defined in the same name scope or module as the referred struct.

This pragma ensures that each character array listed for the specified struct definition is treated as a string type internally within Vortex OpenSplice and operates exactly like a regular string. This allows the character array to be used as a key for the data type, because as far as Vortex OpenSplice is concerned the character array is in fact a string. On the API level (*e.g.*, generated code) the character array is maintained so that applications will be able to use the field as a regular character array as normal. Be aware that listing a character array here does *not* promote the character array to a key of the data type; the regular keylist pragma must still be used for that. In effect this pragma can be used to let any character array be treated as a string internally, although that is not by definition desirable.

When a character array is mapped to a string internally by using the cats pragma, the product behaves as follows:

1. If the character array does not have a '0' terminator, the middleware will add a `\0` terminator internally and then remove it again in the character array that is presented to a subscribing application. In other words, a character array used in combination with the cats pragma does not need to define a `\0` terminator as one of its elements.
2. If the character array does have a `\0` terminator, the middleware will only process the characters up to the first element containing the `\0` character; all other characters are ignored. The middleware will present the character array with the same `\0` terminator to a subscribing application and any array elements following that `\0` terminator will contain `\0` terminators as well; *i.e.*, any array elements following a `\0` element are ignored.

The following table shows some examples using the *cats* pragma for a character array with a size of `4`.

Character array written	*Internal string representation*	*Character array read*
(by publishing application)	(Internal OpenSplice data)	(By subscribing application)
`['a','b','c','d']`	`"abcd"`	`['a','b','c','d']`
`['a','b','c','\0']`	`"abc"`	`['a','b','c','\0']`
`['a','b','\0','d']`	`"ab"`	`['a','b','\0','\0']`

7.2 Bounded strings as character arrays

In some use cases a large number of (relatively small) strings may be used in the data model, and because each string is a reference type, it means that it is not stored inline in the data model but instead as a pointer. This will result in separate allocations for each string (and thus a performance penalty when writing data) and a slight increase in memory usage due to pointers and (memory storage) headers for each string.

The Vortex OpenSplice IDL Pre-Processor features a special pragma called `stac` which can be utilized in such use cases. This pragma enables you to indicate that Vortex OpenSplice should store strings internally as character arrays (but on the API level they are still bounded strings). Because a character array has a fixed size, the pragma `stac` only affects bounded strings. By storing the strings internally as a character array the number of allocations is reduced and less memory is used. This is most effective in a scenario where a typical string has a relatively small size, *i.e.* fewer than 100 characters.

Using the pragma `stac` on bounded strings results in the limitation that those strings can no longer be utilized in queries. It also results in the maximum size of the bounded string to be used each time, therefore the pragma `stac` is less suitable when the string has a large bound and does not always use up the maximum space when filled with data. A bounded string that is also mentioned in the pragma `keylist` can not be listed for pragma `stac`, as transforming those strings to an array would violate the rule that an array can not be a keyfield.

```
#pragma stac <data-type-name> [[!]bounded-string-fieldname]\*
```

The identifier *<data-type-name>* is the identification of a struct definition. The identifier *[[!]bounded-string-fieldname]* is the member of a struct with the type bounded string. The `stac` pragma must be defined in the same name scope or module as the referred struct. If no field names are listed, then all bounded strings will be treated as character arrays internally. If only a subset of the struct members is targeted for transformation then these members can be listed explicitly one by one. Preceeding a field name with a `!` character indicates that the listed member should not be considered for transformation from bounded string to character array.

Member names with and without the `!` character may not be mixed within a `stac` pragma for a specific struct as this has no relevant meaning. This pragma ensures that each bounded string listed for the specified struct definition is treated as a character array type internally within Vortex OpenSplice and operates exactly like a regular bounded string. On the API level (*i.e.* , generated code) the bounded string is maintained so that applications will be able to use the field as a regular bounded string.

CHAPTER

EIGHT

MODES, LANGUAGES AND PROCESSING STEPS

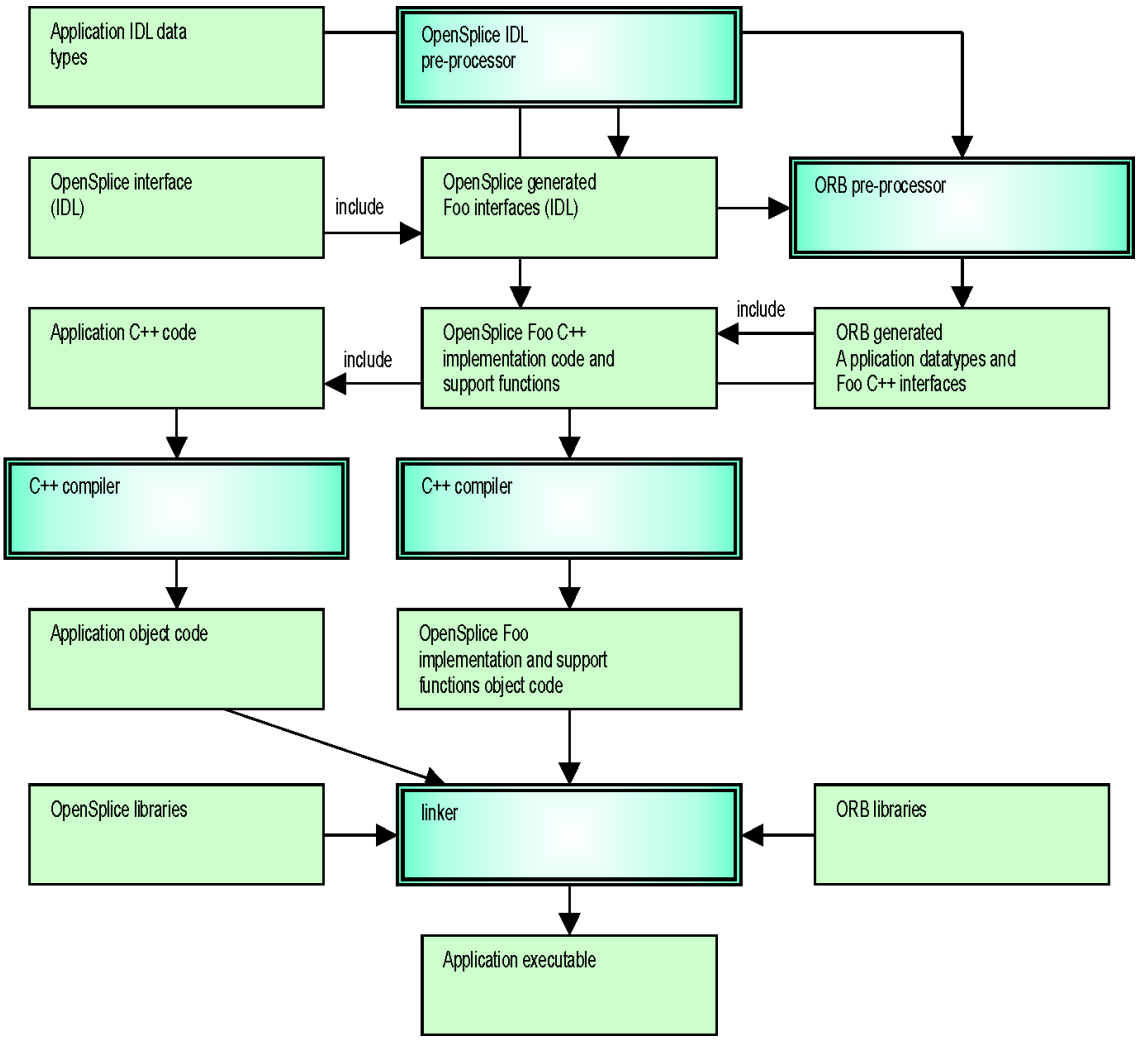
8.1 Integrated C++ ORB

The generic diagram for the ORB integrated C++ context is shown in the diagram *Integrated C++ ORB*.

The Vortex OpenSplice IDL Pre-processor generates IDL code for the specialized *TypeSupport*, *DataReader* and *DataWriter*, as well as C++ implementations and support code. The ORB pre-processor generates from the generated IDL interfaces the C++ specialized interfaces for that specific ORB. These interfaces are included by the application C++ code as well as the Vortex OpenSplice generated specialized C++ implementation code. The application C++ code as well as the specialized C++ implementation code (with the support functions) is compiled into object code and linked together with the applicable Vortex OpenSplice libraries and the ORB libraries.

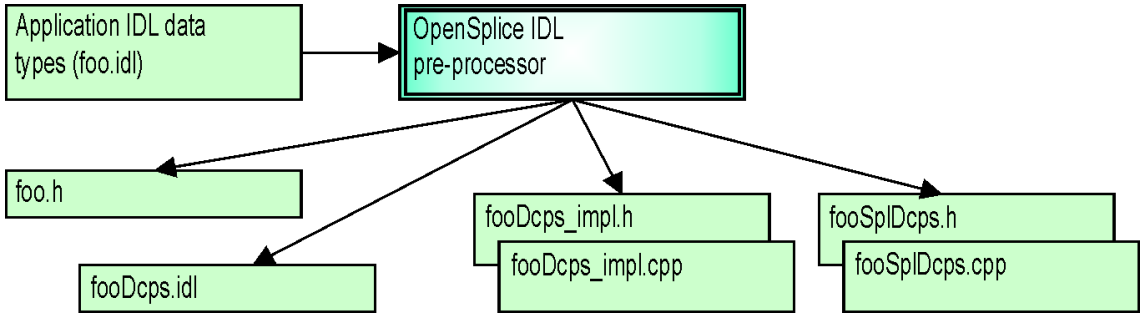
Vortex OpenSplice libraries are provided for linking with OpenFusion TAO. However the source code of the C++ API is also available to build against your own ORB and/or compiler version.

Integrated C++ ORB



The role of the Vortex OpenSplice IDL Pre-processor functionality is expanded in *Integrated C++ ORB Vortex OpenSplice IDL Pre-processor Details*. It shows in more detail which files are generated, given an input file (in this example `foo.idl`).

Integrated C++ ORB Vortex OpenSplice IDL Pre-processor Details



The file `foo.h` is the only file that needs to be included by the application. It includes all files needed by the application to interact with the DCPS interface.

The file `fooDcps.idl` is an IDL definition of the specialized *TypeSupport*, *DataReader* and *DataWriter* interfaces, which will be used to generate ORB-specific C++ interface files.

The `fooDcps_impl.*` files contain the specialized *TypeSupport*, *DataReader* and *DataWriter* implementation classes needed to communicate the type *via* Vortex OpenSplice.

The `fooSplDcps.*` files contain support functions required by Vortex OpenSplice in order to be able to handle the specific data types.

8.2 C++ Standalone

The *C++ standalone* mode provides a Vortex OpenSplice context which does not need an ORB. Vortex OpenSplice resolves all implied IDL-to-C++ language mapping functions and requirements.

The only difference when using the standalone mode is that DDS is used as the naming scope for definitions and functions instead of the CORBA naming scope (the CORBA namespace is still supported, however, for compatibility purposes).

The diagram *C Standalone* is an overview of the artifacts and processing stages related to the C standalone context. For C++ the different stages are equal to the C standalone context. Because there is no ORB involved, all pre-processing is performed by the Vortex OpenSplice IDL Pre-processor. The generated specialized implementations and the application's C++ code must be compiled into object code, plus all objects must be linked with the appropriate Vortex OpenSplice libraries.

8.3 ISOC++

The *ISOC++* mode provides a Vortex OpenSplice context which does not need an ORB. Vortex OpenSplice resolves all implied IDL-to-C++ language mapping functions and requirements. Much like C++ standalone mode, the CORBA naming scope is not used but `C99` types are used in place of `DDS::` types, as specified in the *ISO/IEC C++ language mapping specification*.

8.4 ISOC++2

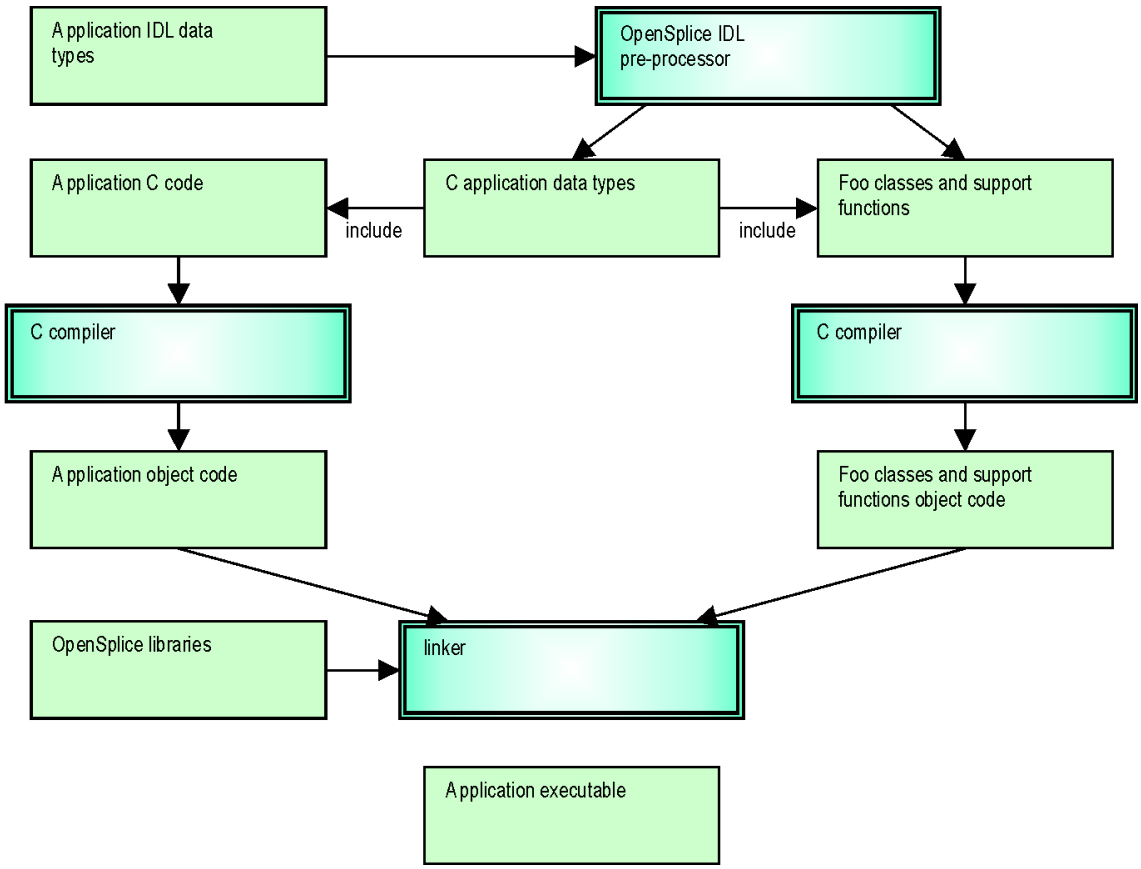
The *ISOC++2* mode provides a Vortex OpenSplice context which does not need an ORB. Vortex OpenSplice resolves all implied IDL-to-C++ language mapping functions and requirements. Much like C++ standalone mode, the CORBA naming scope is not used but `C99` types are used in place of `DDS::` types, as specified in the *ISO/IEC C++ language mapping specification*.

8.5 C Standalone

The *C standalone* mode provides an Vortex OpenSplice context which does not need an ORB. Vortex OpenSplice resolves all implied IDL to C language mapping functions and requirements. The only difference when using the standalone mode is that DDS is used as the naming scope for definitions and functions.

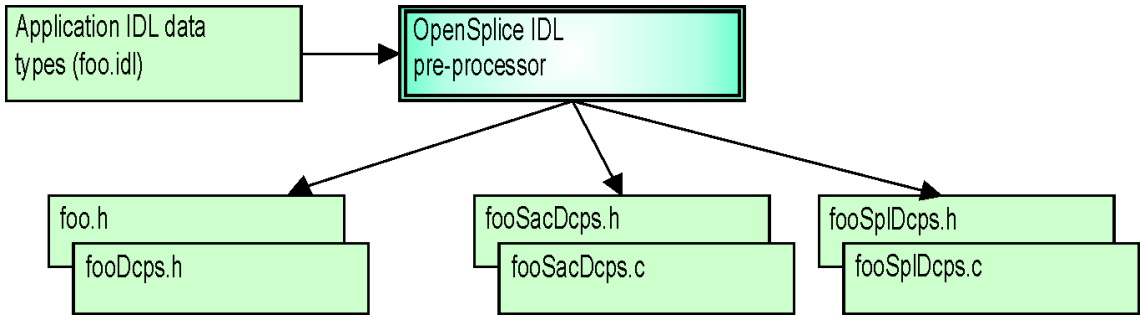
The diagram *C Standalone* shows an overview of the artifacts and processing stages related to the C standalone context. Because there is no ORB involved, all the pre-processing is done by the Vortex OpenSplice IDL Pre-processor. The generated specialized class implementations and the application's C code must be compiled into object code, plus all objects must be linked with the appropriate Vortex OpenSplice libraries.

C Standalone



The role of the Vortex OpenSplice IDL Pre-processor functionality is expanded in the diagram *C Standalone Vortex OpenSplice IDL Pre-processor Details*, providing more detail about the files generated when provided with an input file (`foo.idl` this example).

C Standalone Vortex OpenSplice IDL Pre-processor Details



The file `foo.h` is the only file that needs to be included by the application. It itself includes all necessary files needed by the application in order to interact with the DCPS interface.

The file `fooDcps.h` contains all definitions related to the IDL input file in accordance with the *OMG's IDL-to-C language mapping specification*.

The `fooSacDcps.*` files contain the specialized *TypeSupport*, *DataReader* and *DataWriter* classes needed to communicate the type *via* Vortex OpenSplice.

The `fooSplDcps.*` files contain support functions required by Vortex OpenSplice in order to be able to handle the specific data types.

8.6 C99 Standalone

The *C99 standalone* mode is simular to the *C standalone* mode. The difference is that the *C99 standalone* mode is used to support the C99 version of the C programming language. Except for some small changes in the generated artifacts this mode operates the equal to the *C standalone* mode. See for a description of the processing stages the description in *section C Standalone*.

The difference with the *C standalone* mode is that the primitive types are mapped to the corresponding C99 types. Further bounded strings are mapped to char array's with a upperboundone larger than specified in the idl to allow for the terminating 0 character.

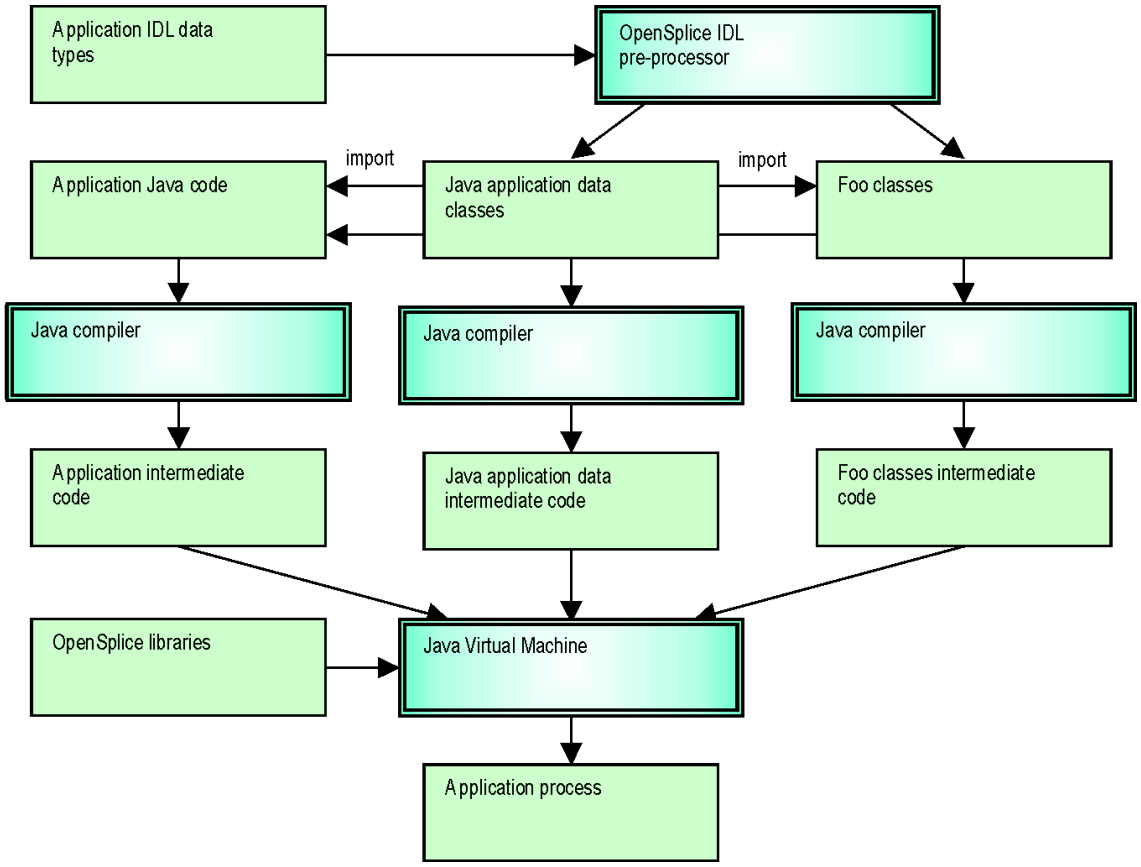
Further an additional file `fooDcps.c` is generated which contains the information to register the type information with Vortex OpenSplice.

8.7 Java Standalone

The *Java standalone* mode provides a Vortex OpenSplice context without the need of an ORB, which still enables portability of application code because all IDL Java language mapping implied functions and requirements are resolved by Vortex OpenSplice.

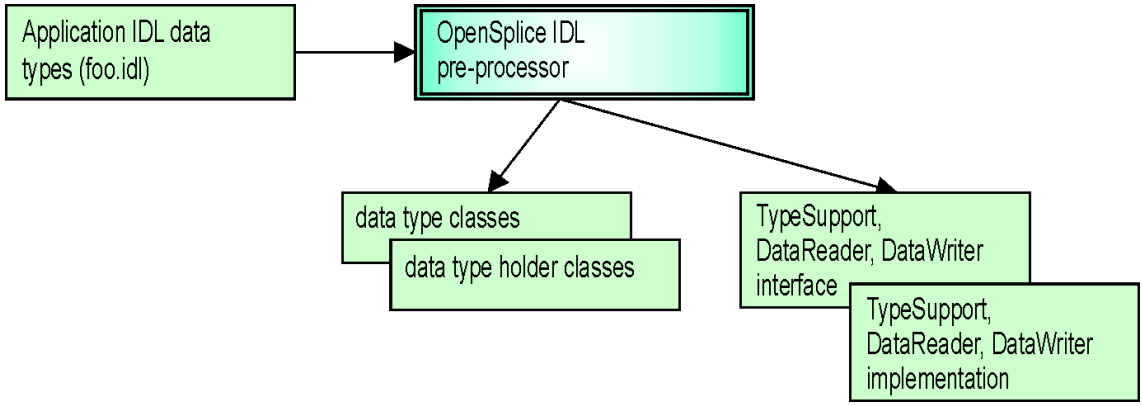
The diagram *Java Standalone* shows an overview of the artifacts and processing stages related to the Java standalone context. The Vortex OpenSplice IDL Pre-processor generates the application data classes from IDL according the language mapping. The Vortex OpenSplice IDL Pre-processor additionally generates classes for the specialized TypeSupport, DataReader and DataWriter interfaces. All generated code must be compiled with the Java compiler as well as the application Java code.

Java Standalone



The role of the Vortex OpenSplice IDL Pre-processor functionality is more magnified in the diagram *Java Standalone OpenSplice IDL Pre-Processor Details*. It shows in more detail which files are generated based upon input file (in this example `foo.idl`).

Java Standalone OpenSplice IDL Pre-Processor Details



8.8 Integrated Java ORB

The *Java CORBA* mode provides a Vortex OpenSplice context for the JacORB ORB. The Vortex OpenSplice IDL Pre-processor generates IDL code for the specialized *TypeSupport*, *DataReader* and *DataWriter*, as well as Java implementations and support code. The ORB pre-processor generates the Java `'Foo'` classes, which must be done manually. These classes are included with the application Java code as well as the Vortex OpenSplice generated specialized Java implementation code. The application Java code as well as the specialized Java implementation code (with the support functions) is compiled into class files and can be used together with the applicable Vortex OpenSplice libraries and the ORB libraries.

The artifacts and processing stages related to the Java CORBA cohabitation context are similar to those of the standalone mode, with one exception: the `'Foo'` classes will not be generated by the Vortex OpenSplice IDL Pre-processor. Instead these classes should be generated by the JacORB IDL Pre-processor.

CHAPTER

NINE

EXTENSIBLE AND DYNAMIC TOPIC TYPES FOR DDS ANNOTATION SUPPORT

The specification defines an annotation language as extension to IDL. Even though this specification has not been implemented in Vortex OpenSplice, its IDL pre-processor is already able to parse these extensions even though it does not generate anything special for them yet. This allows users to write future-proof IDL definitions with annotations already in them.

The specification describes *two* notations for defining annotations in IDL, a prefix and a suffix notation. An annotation type is defined by prefixing a local interface definition with the new token `@Annotation`. The members of these types shall be represented using IDL attributes, as shown in the following example using the prefix notation:

```
1  @Annotation
2  local interface MyAnnotation {
3     attribute long my_annotation_member_1;
4     attribute double my_annotation_member_2;
5  };
```

Alternately and equivalently, an annotation can be defined by suffixing the interface with the new annotation token using *slash-slash-at* (`//@Annotation`) instead, like this:

```
1  local interface MyAnnotation {
2     attribute long my_annotation_member_1;
3     attribute double my_annotation_member_2;
4  }; //@Annotation
```

An annotation can be applied to a type or type member by prefixing it with an *at* sign (`@`) and the name of the annotation type to apply. To specify the values of any members of the annotation type, include them in name=value syntax between parentheses; for example:

```
1  @MyTypeAnnotation
2  struct Gadget {
3     @MyAnnotation(my_annotation_member_1=5,
   my_annotation_member_2=3.4) long my_integer;
4  };
```

Alternately and equivalently, an annotation can also be applied to a type or type member by suffixing it with an annotation type name using *slash-slash-at* (`//@`) instead of the *at* sign by itself; for example:

```
1  struct Gadget {
2  long my_integer;//@MyAnnotation(my_annotation_member_1=5,
   my_annotation_member_2=3.4)
3  }; //@MyTypeAnnotation
```

C++ Java

Please note that the IDL Pre-Processor *only* supports the *suffix* notation when selecting C++ (`-l cpp`, `-l c++`, `-l isocpp`, `-l isoc++`, `-l isocpp2` or `-l isoc++2`) or CORBA-cohabitated java (`-l java -C`) as language.

For other languages both prefix and suffix notations are supported. The *Extensible and Dynamic Topic Types for DDS specification* also defines a number of annotations for use by applications. These types do not appear as annotations at runtime; they exist at runtime only in order to extend the capabilities of IDL. The following annotations have been defined and are accepted by the Vortex OpenSplice IDL pre-processor as well:

- ID
- Optional
- Key
- Shared
- BitBound
- Value
- BitSet
- Nested
- Extensibility
- MustUnderstand
- Verbatim

For more details on built-in annotations and annotations in general please refer to section 7.3.1 of the *OMG Extensible and Dynamic Topic Types for DDS specification*.

CHAPTER

TEN

BUILT-IN DDS DATA TYPES

The Vortex OpenSplice IDL Pre-processor and the Vortex OpenSplice runtime system support the following DDS data types to be used in application IDL definitions:

- `Duration_t`
- `Time_t`

When building C or Java application programs, no special actions have to be taken other than enabling the Vortex OpenSplice IDL Pre-processor built-in DDS data types using the `-o dds-types` option.

For C++, however, attention must be paid to the ORB IDL compiler, which is also involved in the application building process. The ORB IDL compiler is not aware of any DDS data types, so the supported DDS types must be provided by means of inclusion of an IDL file (`dds_dcps.idl`) that defines these types. This file must not be included for the Vortex OpenSplice IDL Pre-processor, which has the type definitions built-in. Therefore `dds_dcps.idl` must be included conditionally. The condition can be controlled *via* the macro definition `OSPL_IDL_COMPILER`, which is defined when the Vortex OpenSplice IDL Pre-processor is invoked, but *not* when the ORB IDL compiler is invoked:

```
#ifndef OSPL_IDL_COMPILER
#include <dds_dcps.idl>
#endif

module example {
  struct example_struct {
    DDS::Time_ttime;
  };
};
```

The ORB IDL compiler must be called *with* the `-I$OSPL_HOME/etc/idlpp` option in order to define the include path for the `dds_dcps.idl` file. The Vortex OpenSplice IDL Pre-processor must be called *without* this option.

CHAPTER

ELEVEN

REFERENCES

The following documents are referred to in the text:

OMG DDS 2004

Object Management Group,
'Data Distribution Service for Real-Time Systems',
Final Adopted Specification, ptc/04-04-12
2004

OMG CORBA v3 2002

Object Management Group,
'The Common Object Request Broker: Architecture and Specification',
Version 3.0, formal/02-06-01
2002

OMG C Language 1999

Object Management Group,
'C Language Mapping Specification',
Version 1.0, formal/99-07-35
1999

OMG C++ Language 2003

Object Management Group,
'C++ Language Mapping Specification',
Version 1.1, formal/03-06-03
2003

OMG Java Language 2002

Object Management Group,
'Java Language Mapping Specification',
Version 1.2, formal/02-08-05
2002

OMG ISO/IEC C++ Language 2013

Object Management Group,
'ISO/IEC C++ 2003 Language DDS PSM',
Version 1.0, formal/2013-11-01
2013

OMG DDS XTYPES 2012

Object Management Group,
'Extensible and Dynamic Topic Types for DDS',
Version 1.0, formal/2012-11-10
2012

CHAPTER

TWELVE

CONTACTS & NOTICES

12.1 Contacts

ADLINK Technology Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA
Tel: +1 781 569 5819

ADLINK Technology Limited
The Edge
5th Avenue
Team Valley
Gateshead
NE11 0XA
UK
Tel: +44 (0)191 497 9900

ADLINK Technology SARL
28 rue Jean Rostand
91400 Orsay
France
Tel: +33 (1) 69 015354

Web: https://www.adlinktech.com/en/data-distribution-service

Contact: https://www.adlinktech.com/en/data-distribution-service

E-mail: ist_info@adlinktech.com

LinkedIn: https://www.linkedin.com/company/79111/

Twitter: https://twitter.com/ADLINKTech_usa

Facebook: https://www.facebook.com/ADLINKTECH

12.2 Notices