OpenSplice DDS

Version 6.x

IDL Pre-processor Guide

PRISMTECH

OpenSplice DDS

IDL Pre-processor Guide

Part Number: OS-IDLP

Doc Issue 27, 20 June 2012

Copyright Notice

Contents

Table of Contents

Preface

About the IDL Pre-processor Guide

The *IDL Pre-processor Guide* describes what the OpenSplice DDS IDL Pre-processor included with the OpenSplice DDS product is and how to use it.

Intended Audience

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

Organisation

Section 1.1, *Introduction*, provides a high-level description and brief introduction of the IDL Pre-processor.

Section 1.2, *Prerequisites*, describes the prerequisites needed to run the pre-processor.

Section 1.3, *IDL Pre-processor Command Line Options*, provides the options which are available for running the pre-processor.

Section 1.4, *OpenSplice DDS Modes and Languages*, provides a summary of OpenSplice's supported modes and languages, as well as an overview of the applicable OpenSplice DDS libraries.

Section 1.5, *IDL Pre-processor Grammar*, shows the IDL grammar that is supported by the OpenSplice DDS IDL Pre-processor.

Section 1.7, *Modes, Languages and Processing steps* describes the steps required for creating programs for each of the modes and languages supported by the Pre-processor.

Section 1.8, *Built-in DDS data types* describes the built-in DDS data types and provides language-specific guidelines on how to use them.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the IDL Pre-processor 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.* XP, 2003, Windows 7) 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, such as *Contacts* on page viii, are 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*** are used to indicate new terms, or emphasise an item.

Sans-serif and **Sans-serif Bold** are used to indicate elements of a Graphical User Interface (GUI) or Integrated Development Environment (IDE), such as an OK button, and sequences of actions, such as selecting **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.

USA Corporate Headquarters

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA

Tel: +1 781 569 5819

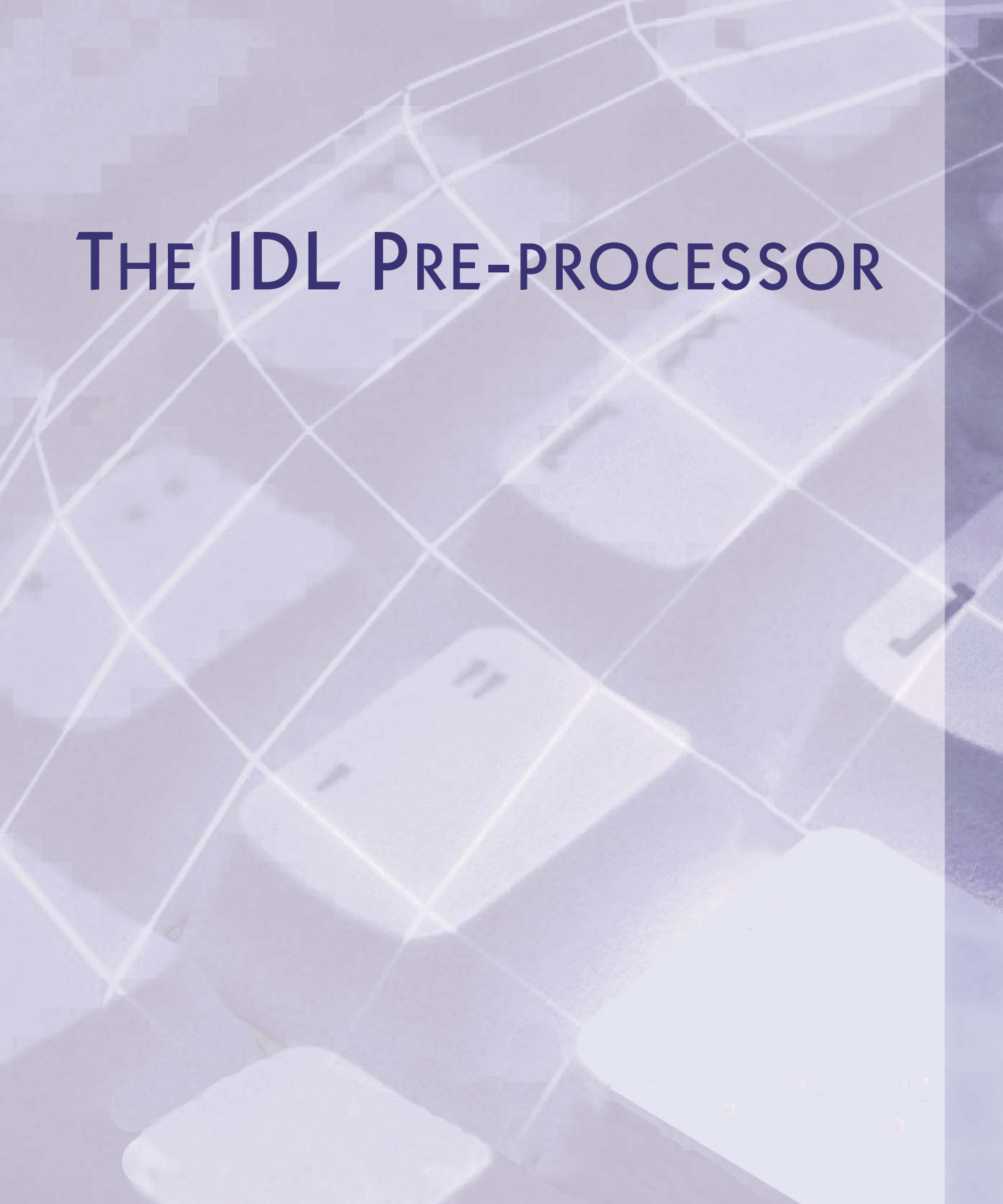
European Head Office

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*
Technical questions: *crc@prismtech.com* (Customer Response Center)
Sales enquiries: *sales@prismtech.com*

The IDL Pre-processor



CHAPTER

1 Description and Use

The OpenSplice DDS 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.

1.1 Introduction

The OpenSplice DDS IDL Pre-processor supports two modes:

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

In a standalone context, OpenSplice DDS 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 OpenSplice DDS only provides the DDS/DCPS related artifacts. The application data type representation provided by the ORB is also used within the OpenSplice DDS context. In this way application data types can be shared between the ORB and OpenSplice DDS within one application program.

The OpenSplice DDS 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.

Figure 1, OpenSplice DDS IDL Pre-processor High Level Processes, on page 4 shows the OpenSplice DDS IDL Pre-processor high-level processing.

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

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

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

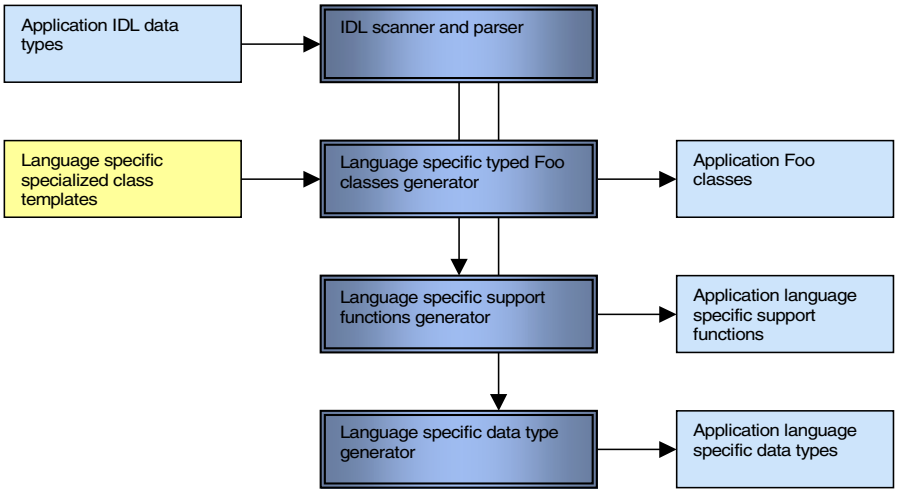


Figure 1 OpenSplice DDS IDL Pre-processor High Level Processes

1.2 Prerequisites

UNIX The OpenSplice DDS environment must be set correctly for UNIX-based platforms before the OpenSplice DDS 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 OpenSplice DDS installation (*<OSPL_HOME>*):

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

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

```
%   idlpp
```

The `idlpp` command line options are describe in Section 1.3, *IDL Pre-processor Command Line Options*, below.

1.3 IDL Pre-processor Command Line Options

The OpenSplice DDS 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) >
[ -j [old]:<new>]
[ -o <dds-types> | <custom-psm> ]
[ -d <output-directory> ]
[ -P <dll_macro_name>[,<header_file>] ]
<filename>
```

These options are described in detail, below. Options shown between angle brackets, < and >, are mandatory. Options shown between square brackets, `[` and `]`, are optional.

-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 build 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.

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

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

- 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:

UNIX — *CCPP/DDS_OpenFusion_1_6_1* for Unix-based platforms.
WIN — *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.

Java - 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.

Java - 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.

C# - 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.

See also Section 1.4, *OpenSplice DDS Modes and Languages*, on page 8 for the supported modes and languages.

Java **`-j [old]:<new>`** - Only applicable to Java. 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 '`.`' characters. A trailing '`.`' character is not required, but may be used.

Example usage: `-j :org.opensplice` (prefixes *all* Java packages).

Example usage: `-j com.opensplice.:org.opensplice.` (substitutes).

`-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 1.8, *Built-in DDS data types*, on page 26.

`-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.

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

WIN **`-P <dll_macro_name>[,<header_file>]`** - This option is only available on Windows platforms. 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 *ex*ported. When *accessing* these operations outside of the DLL, then these external interfaces need to be *im*ported. In case 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 */
```

<filename> - Specifies the IDL input file to process.

1.4 OpenSplice DDS Modes and Languages

The OpenSplice DDS IDL Pre-processor supports two modes:

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

In a standalone context, OpenSplice DDS 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 OpenSplice DDS only provides the DDS/DCPS related artifacts. The application data type representation provided by the ORB is also used within the OpenSplice DDS context. In this way application data types can be shared between the ORB and OpenSplice DDS within one application program.

The languages and modes that OpenSplice DDS supports are listed in *Table 1* below.

Table 1 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
Java	Standalone	dcpssaj.jar	SAJ
Java	ORB integrated	dcpscj.jar	SAJ
C#	Standalone	dcpssacs Assembly.dll	SACS
The language mappings for each language are in accordance with their respective OMG Language Mapping Specifications (see *Bibliography* on page 31).			

1.5 IDL Pre-processor Grammar

The OpenSplice DDS IDL Pre-processor accepts the grammar which complies with the CORBA Specification. The OpenSplice DDS IDL Pre-processor accepts the complete grammar, but 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 OpenSplice DDS IDL Pre-processor are highlighted in ***`bold italic`***. Note that OpenSplice DDS 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*.

```
<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_inheritance_spec> ]
<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_name>          ::= <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>
<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"
<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>
```

```
<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"
<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.
```

1.5.1 Key Definitions

The OpenSplice DDS 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.

1.5.1.1 Supported types for keys

OpenSplice DDS 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 *1.5.1.2* below)

OpenSplice DDS also supports typedef for these types.

1.5.1.2 Character arrays as Keys

By default OpenSplice DDS 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 OpenSplice DDS 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 OpenSplice DDS 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 OpenSplice DDS 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 (by publishing application)	**Internal string representation** (Internal OpenSplice data)	**Character array read** (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']`

1.6 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 OpenSplice DDS IDL Pre-Processor features a special pragma called `stac` which can be utilized in such use cases. This pragma enables you to indicate that OpenSplice DDS 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.* less 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-field-name]` 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 OpenSplice DDS 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.

1.7 Modes, Languages and Processing steps

1.7.1 Integrated C++ ORB

The generic diagram for the ORB integrated C++ context is shown in *Figure 2*. The OpenSplice DDS 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 OpenSplice DDS 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 OpenSplice libraries and the ORB libraries.

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

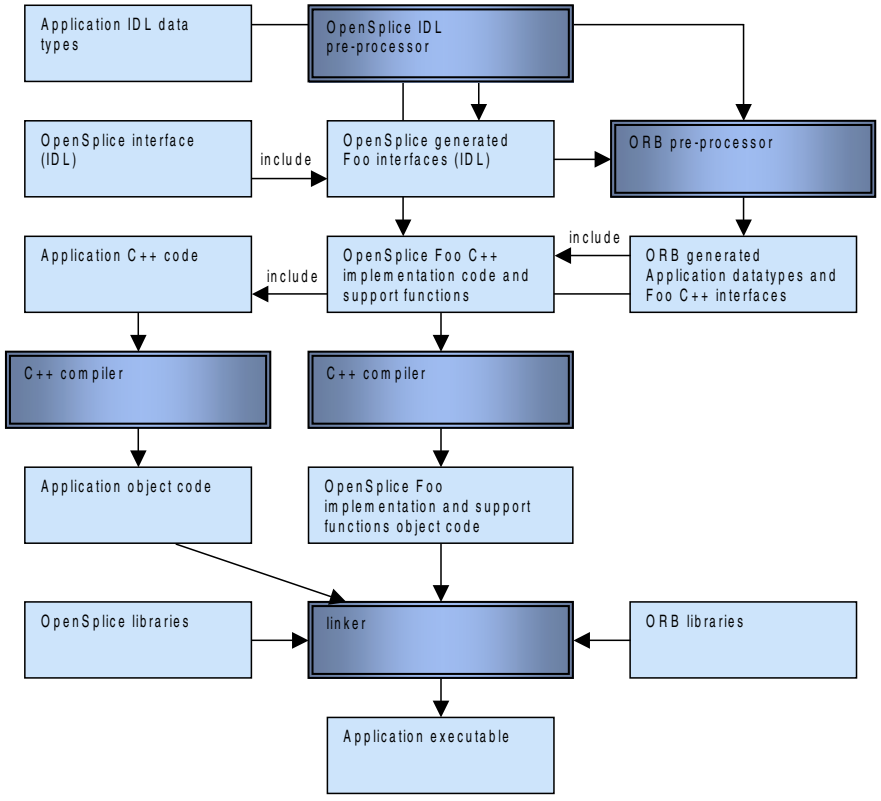


Figure 2 Integrated C++ ORB

The role of the OpenSplice DDS IDL Pre-processor functionality is expanded in *Figure 3*. It shows in more detail which files are generated, given an input file (in this example `foo.idl`).

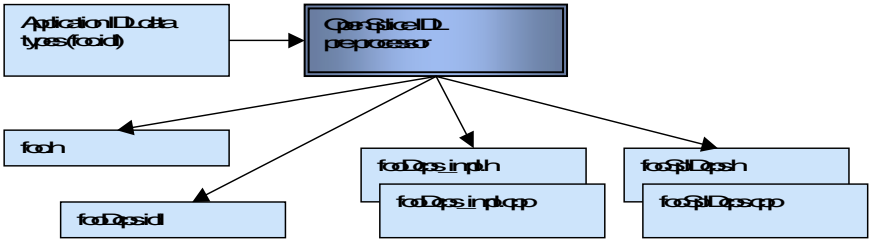


Figure 3 Integrated C++ ORB OpenSplice DDS IDL Pre-processor Details

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 OpenSplice DDS.

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

1.7.2 C++ Standalone

The C++ standalone mode provides an OpenSplice DDS context which does not need an ORB. OpenSplice DDS 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[1].

Figure 4 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 OpenSplice DDS 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 OpenSplice DDS libraries.

1.7.3 C Standalone

The C standalone mode provides an OpenSplice DDS context which does not need an ORB. OpenSplice DDS 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.

Figure 4 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 OpenSplice DDS 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 OpenSplice DDS libraries.

1. The CORBA namespace is still supported, for compatibility purposes.

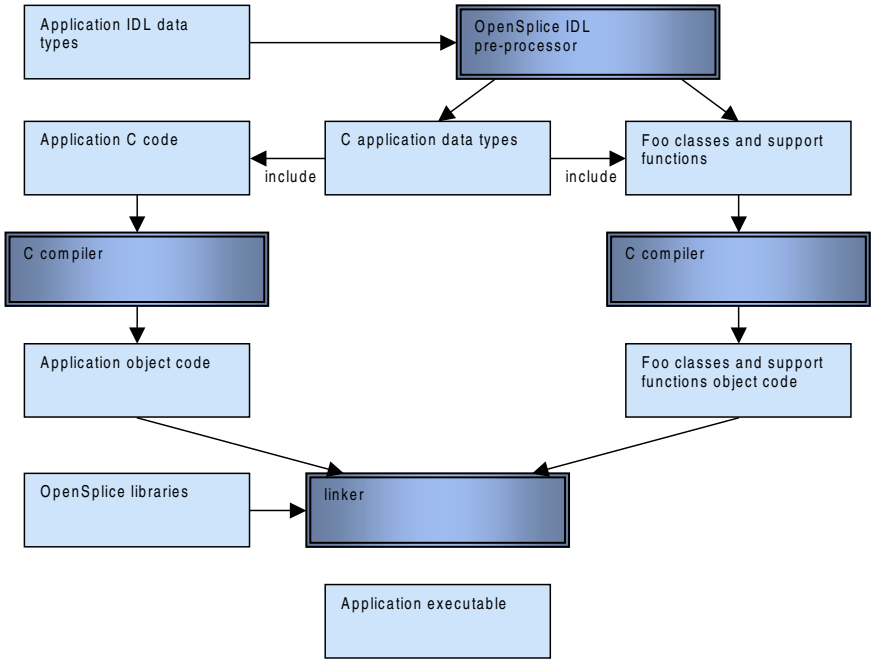


Figure 4 C Standalone

The role of the OpenSplice DDS IDL Pre-processor functionality is expanded in *Figure 5*, providing more detail about the files generated when provided with an input file (`foo.idl` this example).

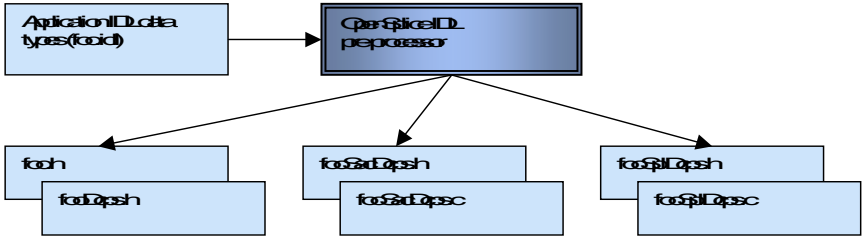


Figure 5 C Standalone OpenSplice DDS IDL Pre-processor Details

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 *C Language Mapping Specification* (IDL to C).

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

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

1.7.4 Java Standalone

The Java standalone mode provides a OpenSplice DDS 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 OpenSplice DDS.

Figure 6 shows an overview of the artifacts and processing stages related to the Java standalone context. The OpenSplice DDS IDL Pre-processor generates the application data classes from IDL according the language mapping. The OpenSplice DDS 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.

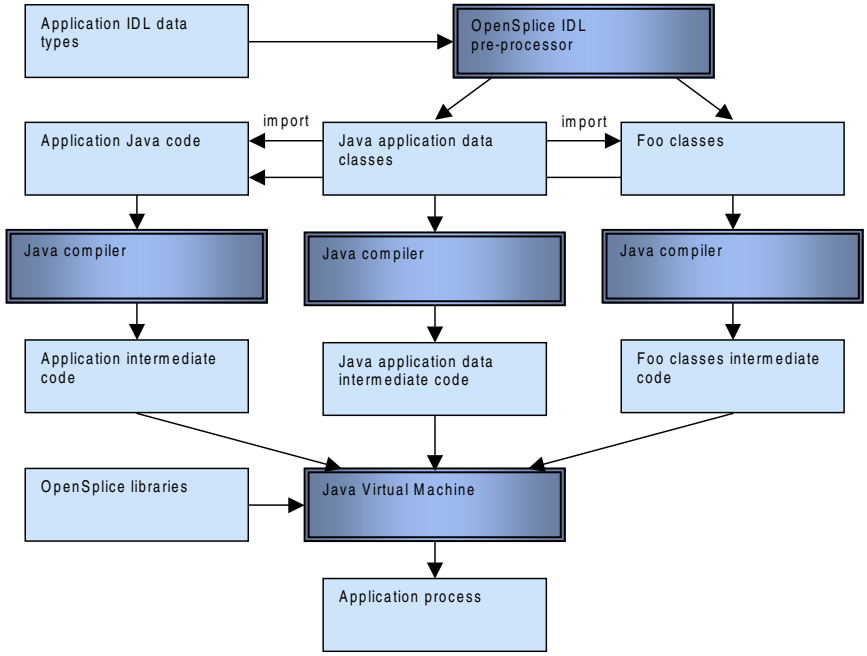


Figure 6 Java Standalone

The role of the OpenSplice DDS IDL Pre-processor functionality is more magnified in *Figure 7*. It shows in more detail which files are generated based upon input file (in this example *foo.idl*).

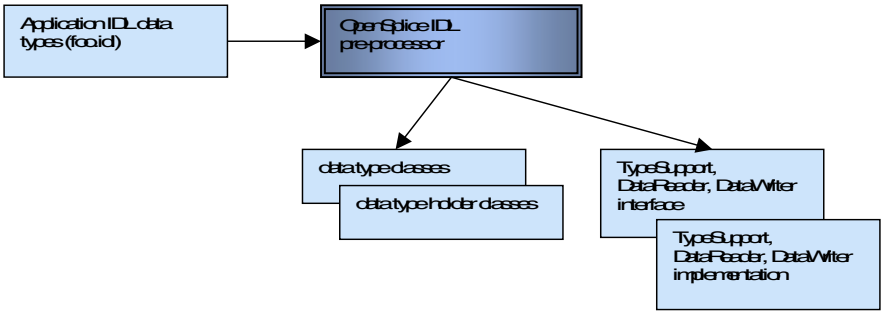


Figure 7 Java Standalone OpenSplice IDL Pre-Processor Details

1.7.5 Integrated Java ORB

The Java CORBA mode provides an OpenSplice DDS context for the JacORB ORB. The OpenSplice DDS 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 OpenSplice DDS 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 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 OpenSplice DDS IDL Pre-processor. Instead these classes should be generated by the JacORB IDL Pre-processor.

1.8 Built-in DDS data types

The OpenSplice DDS IDL Pre-processor and the OpenSplice DDS runtime system supports 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 OpenSplice DDS 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 OpenSplice DDS 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 OpenSplice DDS 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 OpenSplice DDS IDL Pre-processor must be called without this option.

Bibliography

Bibliography

The following documents are referred to in the text:

[1] *Data Distribution Service for Real-Time Systems Specification*, Final Adopted Specification, ptc/04-04-12, Object Management Group (OMG).

[2] *The Common Object Request Broker: Architecture and Specification*, Version 3.0, formal/02-06-01, OMG

[3] *C Language Mapping Specification*, Version 1.0, formal/99-07-35, OMG

[4] *C++ Language Mapping Specification*, Version 1.1, formal/03-06-03, OMG

[5] *Java Language Mapping Specification*, Version 1.2, formal/02-08-05, OMG

Glossary

Glossary

Acronyms

Acronym	Meaning
ASCII	American Standard Code for Information Interchange
BOF	Business Object Facility
CORBA	Common Object Request Broker Architecture
COS	Common Object Services
DCPS	Data Centric Publish Subscribe
DDS	Data Distribution System
EBNF	Extended Backus-Naur Format
IDL	Interface Definition Language
OMG	Object Management Group
ORB	Object Request Broker

Index

Index