OpenSplice DDS

Version 3.4

IDL Pre-processor Guide

PrismTech

OpenSplice DDS

IDL Pre-processor Guide

PrismTech

Part Number: OS-IDLP

Doc Issue 10, 27 May 2008

Copyright Notice

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

Contents

Table of Contents

Preface

The IDL Pre-processor

Preface

About the IDL Pre-processor Guide

The *IDL Pre-processor Guide* describes what the OpenSplice DDS IDL Pre-processor included with the OpenSplice 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 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 IDL Pre-processor.

Section 1.6, *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 1.7, *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. NT, 2000, XP) only.
UNIX	Information applies to Unix based systems (e.g. Solaris) only.
C	C language specific
C++	C++ language specific

Java Java language specific

Hypertext links are shown as *blue italic underlined.*

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

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

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

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

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

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

Arial Bold is used to indicate user related actions, e.g. **File | Save** from a menu.

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

Contacts

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

Corporate Headquarters

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

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

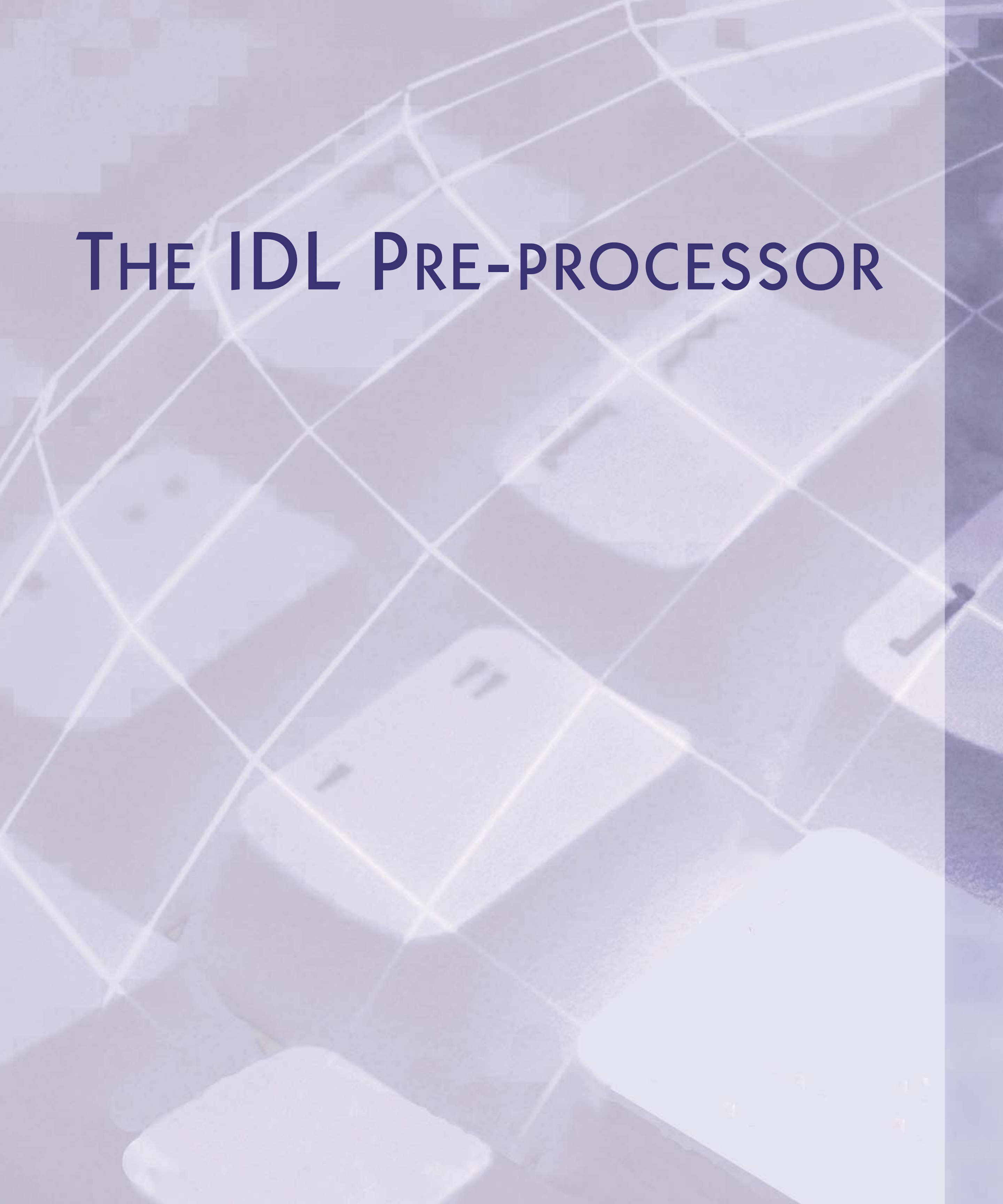
European Head Office

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

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

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

The IDL Pre-processor



CHAPTER

1 Description and Use

The 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.

1.1 Introduction

The OpenSplice 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 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 context. In this way application data types can be shared between the ORB and OpenSplice within one application program.

The 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 IDL Pre-processor "built-in" definitions.

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

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

For the selected language, the 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

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 IDL Pre-processor assumes that the data type will only be used enclosed in other data types.

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

For the standalone context the 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.

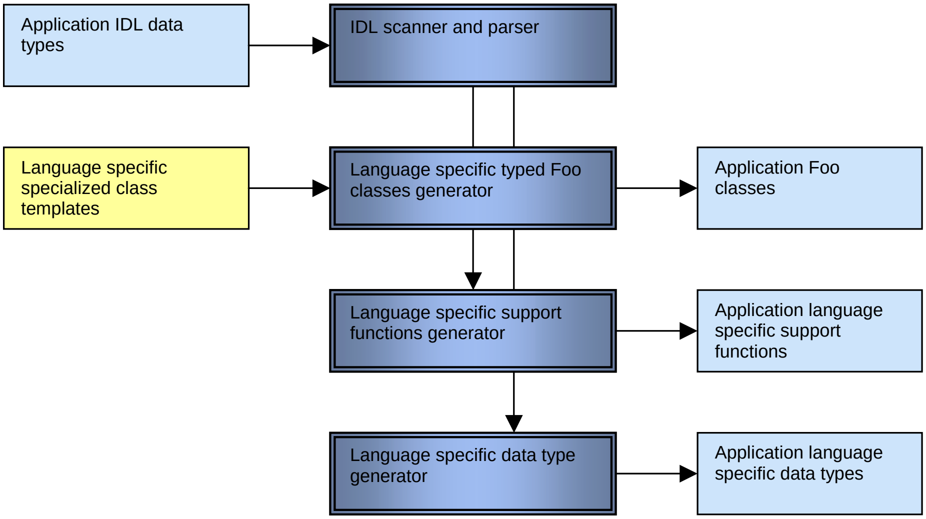

Application IDL data types
IDL scanner and parser
Language specific specialized class templates
Language specific typed Foo classes generator
Application Foo classes
Language specific support functions generator
Application language specific support functions
Language specific data type generator
Application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

Figure 1 OpenSplice 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 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 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 IDL Pre-processor, `idlpp`, can be run with the following command line options:

```
[ -help ]
[ -b ORB-template-path ]
[ -I path ]
[ -D macro[=definition] ]
< -S | -C >
[ -l (c | c++ | cpp | java) ]
[ -o dds-types ]
<filename>
```

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

`-help` - 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.

`-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)` - Selects the target language. Note that the OpenSplice IDL Pre-processor does not support any combination of modes and languages. The default setting of the OpenSplice IDL Pre-processor is *C++*.

For *C*, 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 *C++* (when using the *c++* or *cpp* options) the `OSPL_ORB_PATH` will, by default, be set to:

UNIX for Unix-based platforms - *`CCPP/DDS_OpenFusion_1_4_1`*

WIN for Windows platforms - *`CCPP\DDS_OpenFusion_1_5_1`*

These are the default locations for the IDL to C++ specialized interface for ORB vendor independent template files.

Java For Java, `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. See Section 1.4, *OpenSplice Modes and Languages*, on page 6 for the supported modes and languages.

`-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 1.7, *Built-in DDS data types*, on page 20.

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

1.4 OpenSplice Modes and Languages

The OpenSplice IDL Pre-processor supports two modes:

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

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 only provides the DDS/DCPS related artifacts. The application data type representation provided by the ORB is also used within the OpenSplice context. In this way application data types can be shared between the ORB and OpenSplice 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`	`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.so`	`SAJ`
The language mappings for each language are in accordance with their respective OMG Language Mapping Specifications (see *Bibliography* on page 25).			

1.5 IDL Pre-processor Grammar

The OpenSplice IDL Pre-processor accepts the grammar which complies with the CORBA Specification. The OpenSplice 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 IDL Pre-processor are highlighted in ***bold italic***. Note that OpenSplice does not support all base types that are specified by the OMG.

```
<specification>             ::= <import>* <definition>+
<definition>                ::= <type_dcl> ";" | <const_dcl> ";" | <except_dcl> ";"
                              | <interface> ";" | <module> ";" | <value> ";"
                              | <type_id_dcl> ";" | <type_prefix_dcl> ";"
                              | <event> ";" | <component> ";" | <home_dcl> ";"
<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>
<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" <identifier> "{" <member_list> "}"
<member_list>                 ::= <member>+
<member>                      ::= <type_spec> <declarators> ";"
<union_type>                  ::= "union" <identifier> "switch" "(" <switch_type_spec> ")"
                                  "{" <switch_body> "}"
<switch_type_spec>            ::= <integer_type> | <char_type> | <boolean_type>
                                  | <enum_type> | <scoped_name>
<switch_body>                 ::= <case>+
<case>                        ::= <case_label>+ <element_spec> ";"
<case_label>                  ::= "case" <const_exp> ":" | "default" ":"
<element_spec>                ::= <type_spec> <declarator>
```

```
<enum_type>              ::= "enum" <identifier> "{" <enumerator> { "," <enumerator>}*
                             "}"
<enumerator>             ::= <identifier>
<sequence_type>          ::= "sequence" "<" <simple_type_spec> "," <positive_int_const>
                             ">" | "sequence" "<" <simple_type_spec> ">"
<string_type>            ::= "string" "<" <positive_int_const> ">" | "string"
<wide_string_type>       ::= "wstring" "<" <positive_int_const> ">" | "wstring"
<array_declarator>       ::= <identifier> <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 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.

1.6 Modes, Languages and Processing steps

1.6.1 Integrated C++ ORB

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

i OpenSplice libraries are provided for linking with TAO OpenFusion.

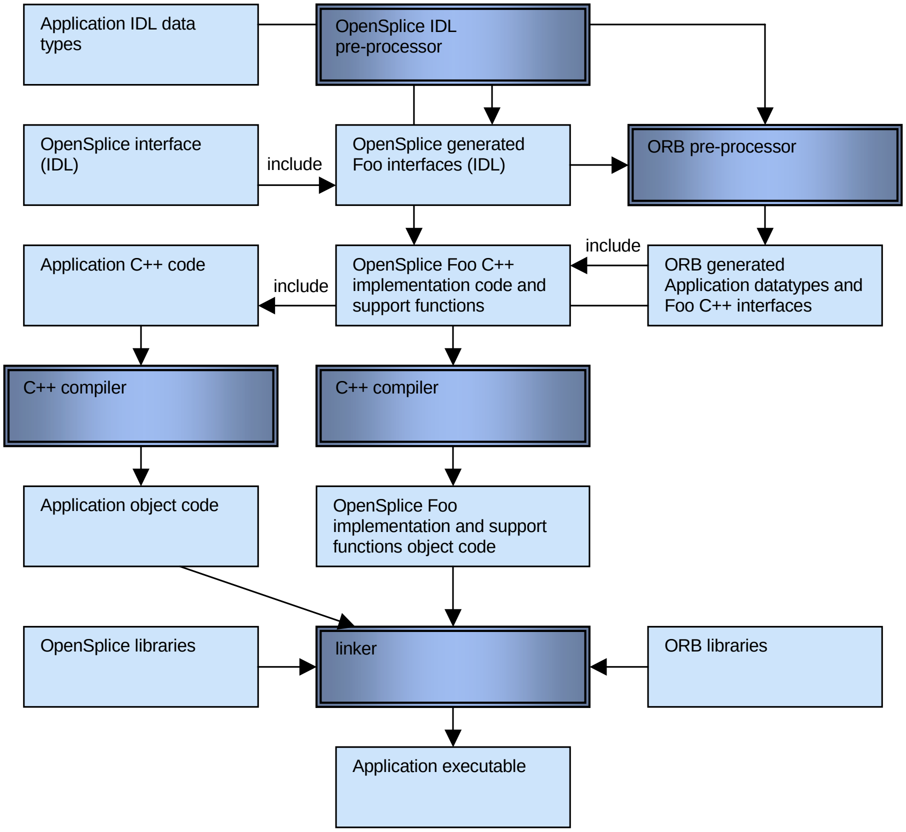

Application IDL data types
OpenSplice IDL pre-processor
OpenSplice interface (IDL)
include
OpenSplice generated Foo interfaces (IDL)
ORB pre-processor
include
Application C++ code
include
OpenSplice Foo C++ implementation code and support functions
ORB generated Application datatypes and Foo C++ interfaces
C++ compiler
C++ compiler
Application object code
OpenSplice Foo implementation and support functions object code
OpenSplice libraries
linker
ORB libraries
Application executable

Figure 2 Integrated C++ ORB

The role of the OpenSplice 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`*).

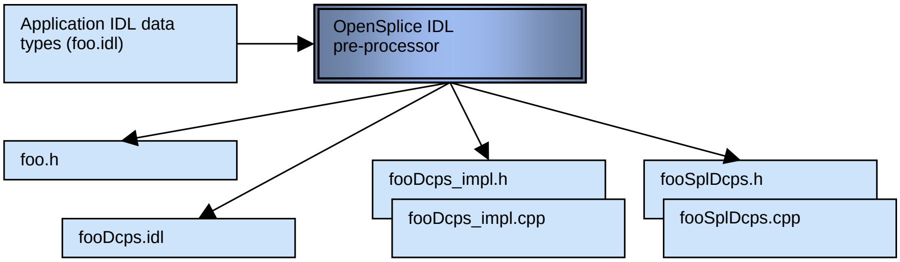


Figure 3 Integrated C++ ORB OpenSplice IDL Pre-processor Details

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.

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

1.6.2 C++ Standalone

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

1.6.3 C Standalone

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

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 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 libraries.

[1] Although for compatibility purposes, the CORBA namespace is still supported.

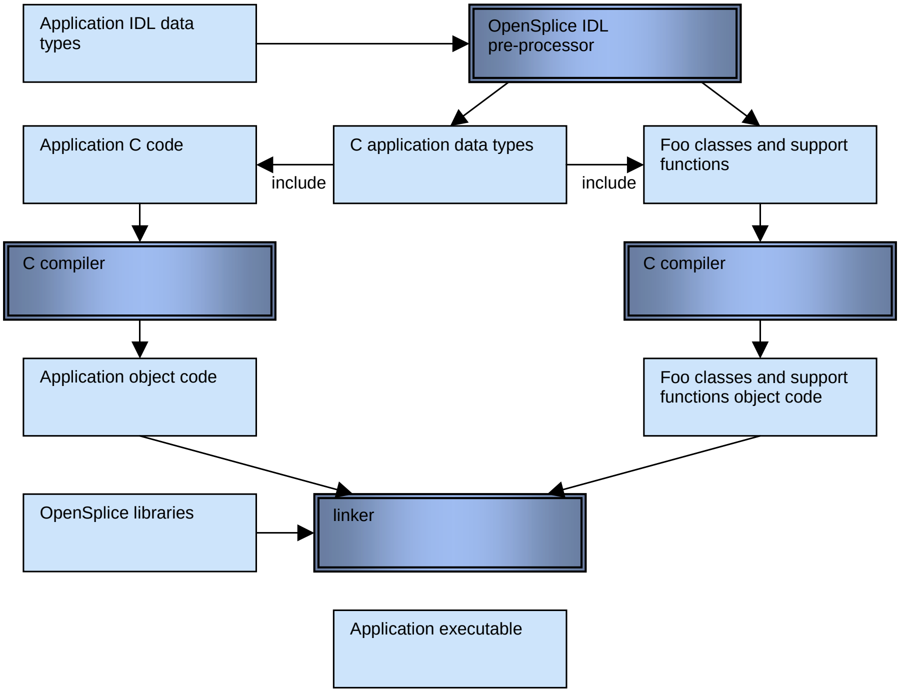


Figure 4 C Standalone

The role of the OpenSplice 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).

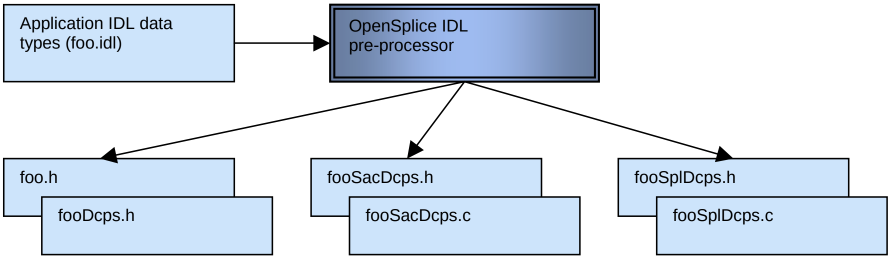

Application IDL data types (foo.idl)
OpenSplice IDL pre-processor
foo.h
fooDcps.h
fooSacDcps.h
fooSacDcps.c
fooSplDcps.h
fooSplDcps.c

Figure 5 C Standalone 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 *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.

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

1.6.4 Java Standalone

The Java standalone mode provides a 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 OpenSplice.

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

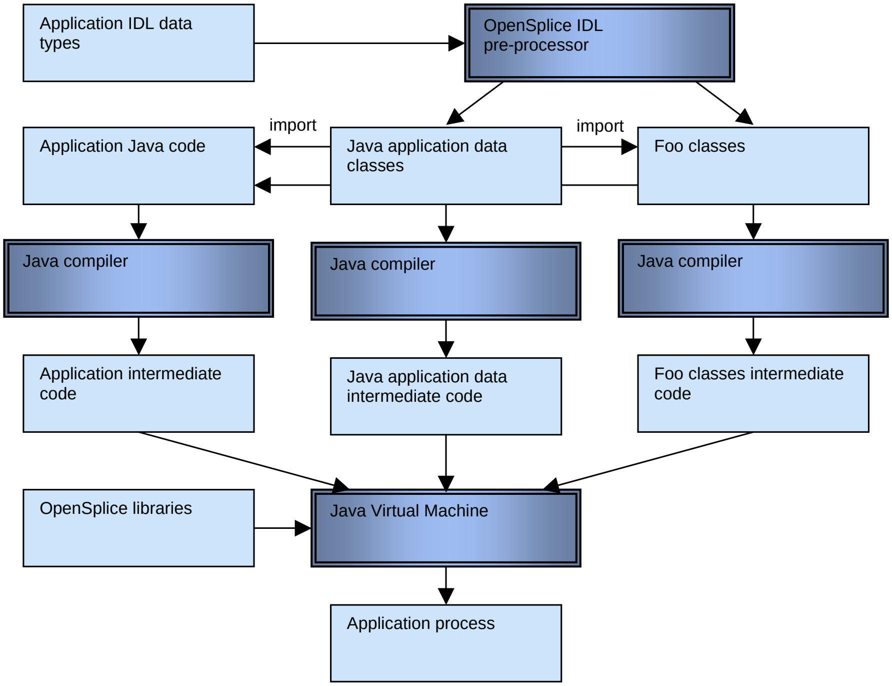

Application IDL data types
OpenSplice IDL pre-processor
import
import
Application Java code
Java application data classes
Foo classes
Java compiler
Java compiler
Java compiler
Application intermediate code
Java application data intermediate code
Foo classes intermediate code
OpenSplice libraries
Java Virtual Machine
Application process

Figure 6 Java Standalone

The role of the OpenSplice 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").

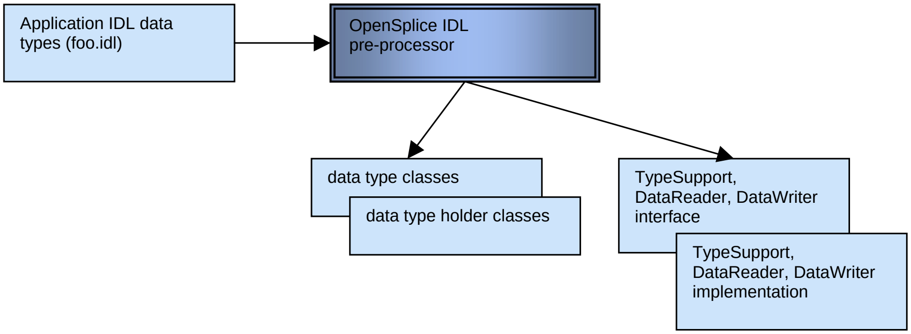

Application IDL data types (foo.idl)
OpenSplice IDL pre-processor
data type classes
data type holder classes
TypeSupport, DataReader, DataWriter interface
TypeSupport, DataReader, DataWriter implementation

Figure 7 Java Standalone OpenSplice IDL Pre-Processor Details

1.7 Built-in DDS data types

The OpenSplice IDL Pre-processor and the OpenSplice 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 IDL Pre-processor built-in DDS data types using the *`-o dds-types`* option.

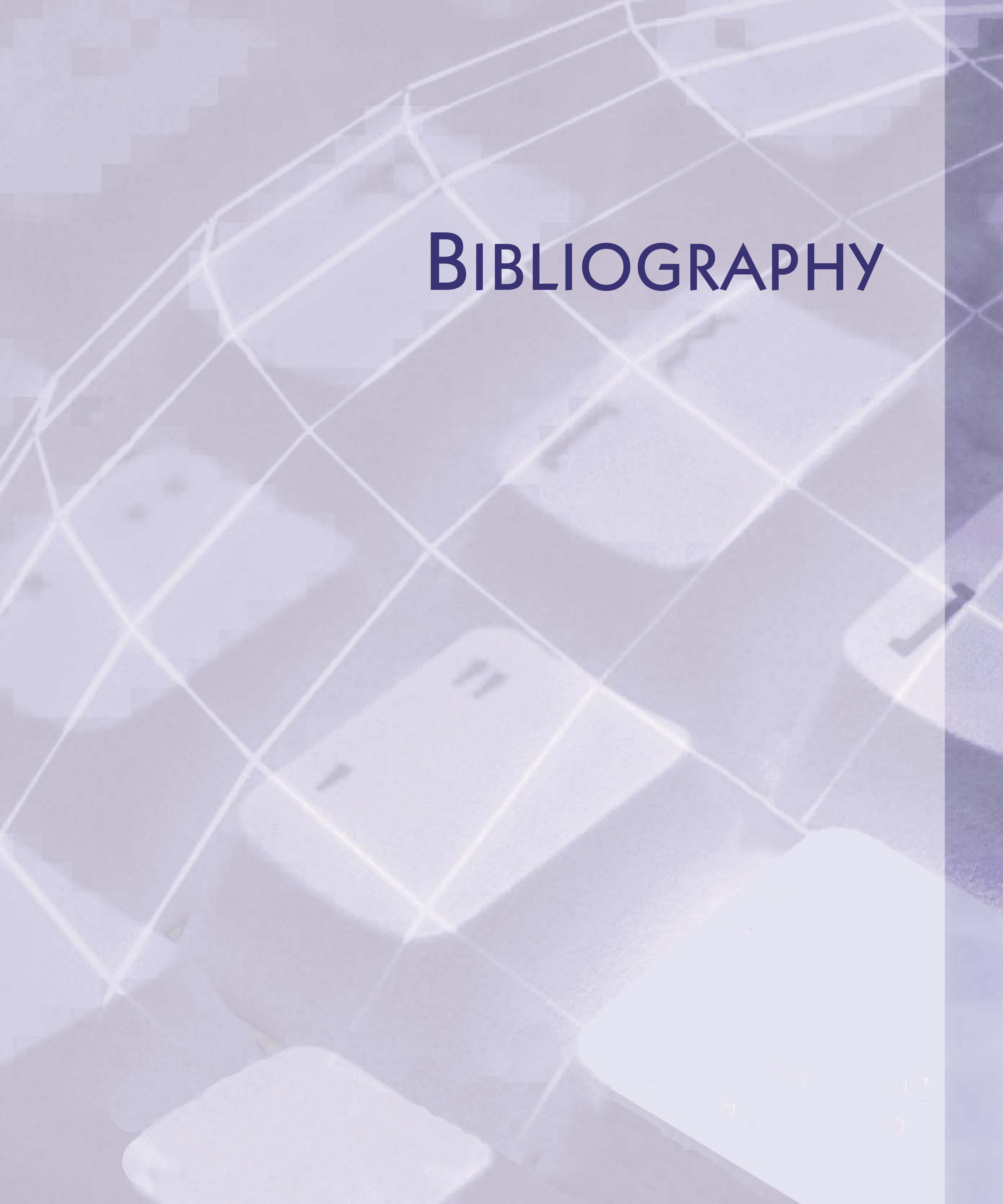
For C++ however attention must be paid to the ORB IDL compiler which is also involved in the application building process. The ORB IDL compiler is not aware of any DDS data types. Because of this, the supported DDS types must be provided by means of inclusion of an IDL file (dds_dcps.idl) that defines these types. This file must however not be included for the 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 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 {
      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 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