mindc User Guide

version 2.0-alpha10

mindc User Guide : version 2.0-alpha10

Table of Contents

List of Examples

Chapter 1. Introduction

An introduction paragraph

Chapter 2. Architecture Description Language

The MIND toolset uses an Architecture Description Language (ADL) for capturing the architecture of an application. In a nutshell, the ADL language provides first-level constructs for defining components in terms of the interfaces they require and provide and to specify their implementations either by referencing a set of C-based implementation files or by assembling a set of sub-components.

An architecture definition defined in ADL may be under three forms :

Primitive component definition	defines a component whose implementation is provided by a set of C-based files.
Composite component definition	defines a component whose implementation is assembled from a set of sub-components which are bound together.
Component type definition	defines a component's abstract architecture in terms of the interfaces it requires and provides. The purpose of a component type definition is the specification of abstract architectures that can be reused with different concrete implementations.

Example 2.1. Primitive component definition

```
1  primitive foo.bar.MyPrimitiveComponent {
2
3    provides foo.Itf1 as aProvidedInterface;
4    provides foo.Itf2 as anotherProvidedInterface;
5    requires foo.bar.Itf3 as aRequiredInterface;
6
7    source myImplementation.c;
8  }
```

In the above primitive component definition:

- Line 3 specifies that the component provides an interface whose name is `aProvidedInterface` and whose signature is `foo.Itf1`. The signature of an interface refers to an interface definition which is captured in a separate IDL file. For further information about the IDL, see Chapter 3, *Interface definition language*;
- Line 4 specifies that the component provides a second interface whose name is `anotherProvidedInterface` and whose signature is `foo.Itf2`;
- Line 5 specifies that the component requires an interface whose name is `aRequiredInterface` and whose signature is `foo.bar.Itf3`;
- Finally, line 7 specifies that the implementation of this primitive component is provided in a file called `myImplementation.c`. For further information about the component implementation in C language, see Chapter 4, *Component Programming Language* .

Example 2.2. Composite component definition

```
 1 composite foo.bar.MyCompositeComponent {
 2
 3   provides foo.Itf1 as itf1;
 4   requires foo.bar.Itf3 as itf2;
 5
 6   contains foo.bar.MyPrimitiveComponent as aSubComponent;
 7   contains foo.YourComponent as anotherSubComponent;
 8
 9   binds anotherSubComponent.requiredItf1
10      to aSubComponent.anotherProvidedInterface;
11
12   binds this.itf1 to aSubComponent.aProvidedInterface;
13 }
```

In the above composite component definition:

- Line 6 specifies that the defined composite component contains a sub-component whose name is `aSubComponent` which is defined in a separate ADL definition called `foo.bar.MyPrimitiveComponent`. For further details about sub-component declarations, see Section 2.5, “Sub-component declarations”;
- Line 7 specifies that the defined composite component contains a second sub-component called `anotherSubComponent` which is defined in a separate ADL definition called `foo.YourComponent`;
- Line 9 and 10 specify that the interface `requiredItf1` of the sub-component called `anotherSubComponent` is bound to the interface `anotherProvidedInterface` of the sub-component called `aSubComponent`. For more information about binding declarations see Section 2.6, “Binding declarations”;
- Finally, line 12 specifies that the interface `itf1` of the defined composite component is bound to the interface `aProvidedInterface` of the sub-component called `aSubComponent`. In this binding statement, the `"this"` keyword designates the defined composite component.

2.1. ADL Files

ADL definitions are written in ADL files. An ADL file contains one and only one *ADL definition* with a unique *fully-qualified-name*. The name of an ADL file must correspond to the simple-name of the definition it contains suffixed by `'.adl'`. The file must be located in a directory that corresponds to the package-name of the definition. For example, a definition whose fully-qualified-name is `foo.bar.MyComponent` must be located in a file `foo/bar/MyComponent.adl`

2.2. Interface declarations

Interfaces are the acces points of a component. Components may have two kinds of interfaces. That is, *provided interfaces* (a.k.a. server interfaces) are implemented by the component whereas *required interfaces* (a.k.a. client interfaces) may be invoked by the component. An interface has a *signature* that is defined in the IDL language (see Chapter 3, *Interface definition language*) and a *name* that must be unique among all the interfaces of the definition (either provided or required).

Provided interfaces are declared with the `"provides"` keyword followed by its signature and its name separated by the `"as"` keyword. Similarly, *required* interfaces are declared with the `"requires"` keyword followed by its signature and its name separated by the `"as"` keyword.

Example 2.3. Simple Interface declaration

```
1  provides foo.Itf1 as aProvidedInterface;
2  provides foo.Itf1 as anotherProvidedInterface;
3  requires foo.bar.Itf3 as aRequiredInterface;
```

In the above code excerpt:

- Line 1 declares a provided interface whose name is `aProvidedInterface` and whose signature is defined in the IDL definition called `foo.Itf1`.
- Line 2 declares a second provided interface interface whose name is `anotherProvidedInterface` and whose signature is `foo.Itf1`. Note that a given component may provide and/or require multiple interfaces with the same signature with different names.
- Line 3 declares that the component requires an interface whose name is `aRequiredInterface` and whose signature is `foo.bar.Itf3`.

An interface may be tagged as *optional*. If a required interface is tagged as optional, the component requiring this interface is expected to be functional even if this interface is not bound. On the other hand, if a provided interface is tagged as optional, the component may not actually provide the implemenation of this interface. Therefore, only optional required interfaces can be bound to optional provided interfaces. This constaint is discussed more in details in Section 2.6, “Binding declarations”. To specify that an interface is optional, the `"optional"` keyword must be added after the `"provides"` or `"requires"` keywords in the interface declaration.

Note

Optional provided interfaces are not fully supported in the current mindc toolchain.

Example 2.4. Optional interface declaration

```
provides optional foo.Itf1 as aProvidedInterface;
requires optional foo.bar.Itf3 as aRequiredInterface;
```

An interface may also be a *collection* interface. A collection interface may be considered as an array of interfaces of the same type, which can be accessed using an index operator. A required collection interface can be bound to several provided interfaces (i.e. each element in the array may be connected to a different provided interface). A provided collection interface allows supporting distinct implementations of the same interface type within the same component. To specify that an interface is actually a collection interface, the number of interfaces that are collected together must be defined between square brackets (i.e. `"["` SIZE `"]"`) right after the name of the interface, similarly to array variable definitions in C.

Note

The current mindc toolchain does not support the collection interfaces where the size is not statically specified in the ADL definition.

Example 2.5. Collection interface declaration

```
provides foo.Itf1 as aProvidedInterface[4];
requires foo.bar.Itf3 as aRequiredInterface[10];
```

An interface can be both optional and collection. In the case of a provided interface, being both collection and optional means the component may not provide some (or any) implementations of the interface. On the other hand, in the case of a required interface, being both collection and optional means that some (or all) of the interfaces may not be bound whereas the component is still suppsed to be functional.

2.3. Implementation declarations

Primitive components are implemented in C language using some macros The ADL definition of primitive components must specify the location of the source files providing the implementation of the component.

There are two constructs in the ADL for specifying the implementation of a primitive component. The first one is used for specifying the declaratrion of the *private-data* of the component. The second one is used for specifying the set of *implementation files* of the component. Further information about the syntactic extensions to the C language that need to be respected in those C implementation files are discussed in detail in Chapter 4, *Component Programming Language* . The rest of this section focuses only on ADL related aspects.

2.3.1. Private data definition

The specification of component's private-data may be done under two different forms using the ADL:

1. The private-data structure definition can be put in a separate header file, that is referenced in the ADL definition using the `"data"` keyword followed by the path to this header file;
2. Or, the private-data structure definition can be inlined directly in the ADL definition in between double curly braces (`"{{"` and `"}}"`) following the `"data"` keyword;

Warning

Source-code inlined in the ADL file is not parsed by the mindc toolchain. It is escaped by the double curly brace (`"{{"` and `"}}"`). Please make sure that consecutive curly braces in the inline source-code are separated by at least one whitespace character.

A primitive component definition may have a maximum of one private-data definition. On the other hand, composite and type definitions cannot specify private-data.

When the private-data structure is defined in a separate header file, the latter should contains only the definition of the private-data structure. It is not necessary to include it in the implememtation files (this is done automatically by the generated code). Moreorver it must not be included by implementations of other components.

See Section 4.1, “Component's private-data declaration” for details on the syntax to be used to define component's private data.

Example 2.6. Private data definition in a separate header file

```
data myComponentData.h;
```

The above code excerpt specifies that the component's private-data are defined in the file called `myComponentData.h`. The latter file should look-like :

```
#ifndef MY_COMPONENT_DATA_H
#define MY_COMPONENT_DATA_H

struct {
  int a, b;
  char c[10];
} PRIVATE;

#endif
```

Example 2.7. Private data definition inlined in the ADL

```
data {{
  struct {
    int a, b;
    char c[10];
  } PRIVATE;
}};
```

The above code excerpt makes an inlined definition of private-data structure directly in the ADL file.

2.3.2. Component implementation declaration

The implementation of a primitive component (i.e. the implementation of its provided interfaces) is done using the C language. For convenience reasons, the implementation of a primitive component may be split into multiple source files. Implementation files must be referenced in the ADL definition of primitive components to make sure the MIND toolset integrates those files with the generated glue code. The declaration of implementation files is done using the `"source"` keyword followed by the path of the source file.

Similarly to private data structure definition, the implementation code may also be inlined in the ADL using double curly braces(`"{{"` and `"}}"`) following the `"source"` keyword.

Example 2.8. Component implementation in a separate C file

```
source myComponentImpl.c;
```

In the above code excerpt, specifies that implementation of the component is located in the `myComponentImpl.c` file. The latter file should look-like :

```
#include <stdio.h>

int METH(adderItf, add) (int a, int b) {
  printf("in adderItf.add\n");
  return a + b;
}
```

Example 2.9. Component implementation inlined in the ADL

```
source {{
  #include <stdio.h>

  int METH(adderItf, add) (int a, int b) {
    printf("in adderItf.add\n");
    return a + b;
  }
}};
```

In the above code excerpt, the implementation of the primitive component is inlined in the ADL file.

Moreover, the `"source"` keyword may be followed by a path to a pre-compiled file (`".o"`, `".a"`, `".so"`, or `.dll` file). In that case, the source file will only be added on the linker command. This is particularly usefull to define a component that wraps a pre-compiled C library.

2.4. Attribute declarations

Attributes are component's data that are declared and initialized in the ADL. This is very useful feature to set some initial parameters for a component (e.g. size of a buffer component). Moreover, if the component implements the *attribute-controller* interface (see Section 6.2.5, “The `attribute-controller`” for more detail), the attributes may also be read and modified at runtime by other components.

Attribute declarations are done using the `"attribute"` keyword followed by a type, a name and optionnaly an initial value.

Note

Currently only type `"int"` is supported in the mindc toolchain.

The initial value of an attribute can be a literal integer (which may optionally be signed with `"+"` or `"-"`) written in decimal, hexadecimal or octal base. The initial value of an attribute can also be a reference to a *definition parameter* as described in Section 2.9, “Definition parameters”.

Example 2.10. Attribute declarations

```
1  attribute int attr1 = -2;
2  attribute int attr2 = 0xff43dc5;
3  attribute int attr3 = myDefinitionParam;
```

In the above code excerpt:

- Line 1 declares an attribute called `attr1` that is initialized to `-2`;
- Line 2 declares another attribute called `attr2` that is initialized to an hexadecimal value;
- Line 3 declares an attribute called `attr3` that is initialized using the definition parameter called `myDefinitionParam`.

2.5. Sub-component declarations

The implementation of a composite component, i.e. the implementation of its provided interfaces, is done by composing a set of sub-components which are bound together in a consistent way. For that purpose, the ADL definition of a composite component may declare a list of sub-components which make part of its implementation.

A sub component is declared using the `"contains"` keyword followed by a reference to an ADL definition and its local name separated by the `"as"` keyword. The local name associated to a sub-component must be unique within the encapsulating composite definition.

Example 2.11. Sub-component declaration

```
1  contains foo.bar.MyPrimitiveComponent as aSubComponent;
2  contains foo.YourComponent as anotherSubComponent;
```

Note

Line 1 declares a sub-component called `aSubComponent` that is an instance of the definition `foo.bar.MyPrimitiveComponent`

2.6. Binding declarations

Components' interfaces must be bound together in order to establish a communication between them. A binding (in its simplest form) is a point-to-point connection that links a required interface (the *client side* of the binding) to a provided interface (the *server side* of the binding). Composite component definitions declare a list of bindings to assemble its sub-components together in a consistent way.

A binding is specified using the `"binds"` keyword followed by the specification of the client side and the server-side which are spearated by the `"to"` keyword. Both client and server sides of a binding are specified with the local-name of the sub-component and the name of the interface of this sub-component separated by a dot (`"."`). The special `"this"` keyword can be used to refer to the encapsulating composite component definition. In the case of collection interfaces, the binding declaration may also specify the indexes of the interfaces to be bound. In this case, the index is specified in between square brackets following the interface name.

Example 2.12. Binding declarations

```
1  binds subComp1.itf1 to subComp2.itf4;
2  binds subComp1.itf2[3] to subComp2.itf2;
3  binds this.itf1 to subComp1.itf3;
4  binds subComp2.itf1 to this.itf2;
```

In the above code excerpt:

- Line 1 declares a binding from the interface `itf1` of the sub-component `subComp1` to the interface `itf4` of the sub-component `subComp2`;
- Line 2 declares a binding from the forth interface of the `itf2` collection interface of the `subComp1` sub-component. Note that the collection interfaces indexes start at zero;
- Line 3 and 4 declare two bindings designating respectively a required and a provided interface of the encapsulating composite component.

TODO detail rules of valid bindings

- Multiple client interfaces can be connected to a single server interface, whereas a single client interface cannot be connected to multiple server interfaces. A collection client interface is needed for that purpose.

2.7. Import statements

Import statements allow shortening the references to other ADL or IDL definitions using their simple-names rather than their *fully-qualified-name*. This feature is directly inspired from the Java programming language.

Import statements must be placed at the very begining of ADL files, preceeding any `"primitive"`, `"composite"` or `"type"` keywords. An import statement is composed of the `"import"` keyword followed by a fully-qualified-name or a package-name itself followed by `".*"`.

The first form (i.e. fully-qualified-name) allows to reference the specified ADL or IDL definition using its simple name.

The second form (i.e. package-name followed by `".*"`) allows to reference any ADL or IDL definition belonging to specified package using their simple names.

ADL or IDL definitions beloning to the same package as the current definition are implicitly imported and can be referenced with there simple-name

Example 2.13. Import statements

The following example can be refactored to use import

```
 1  composite foo.bar.MyCompositeComponent {
 2
 3    provides foo.Itf1 as itf1;
 4    requires foo.bar.Itf3 as itf2;
 5
 6    contains foo.bar.MyPrimitiveComponent as aSubComponent;
 7    contains foo.YourComponent as anotherSubComponent;
 8
 9    ...
10  }
```

Which gives :

```
 1  import foo.bar.Itf3;
 2  import foo.*;
 3
 4  composite foo.bar.MyCompositeComponent {
 5
 6    provides Itf1 as itf1;
 7    requires Itf3 as itf2;
 8
 9    contains MyPrimitiveComponent as aSubComponent;
10    contains YourComponent as anotherSubComponent;
11
12    ...
13  }
```

In the above code excerpt:

- Line 1 imports the IDL definition called `foo.bar.Itf3` which allows to shorten the declaration of the `itf2` interface at line 7 using the simple name (i.e. `Itf3`) rather than its fully qualified name (i.e. `foo.bar.Itf3`).
- Line 2 imports the whole package called `foo`. This allows to shorten the references at line 6 and 10.
- At line 9, the ADL definition called `foo.bar.MyPrimitiveComponent` can be referenced by using its simple name since it is in the same package as the encapsulating (current) definition (`foo.bar.MyCompositeComponent`).
- The import statement at line 1 may be considered as useless since it imports an IDL definition which is already in the same package as the current definition. Nevertheless, such an import statement may be usefull in some cases to explicitly import an ADL or IDL definition in order to avoid name preemption that may occur. For instance, if another IDL whose simple name is `Itf3` is defined in the `foo` package, then the import statement at line 2 would import the latter and therefore `Itf3` would be resolved in `foo.Itf3` rather than in `foo.bar.Itf3`. For more information about the simple-name resolution rules, please refer to Section 2.7.1, “Resolution of simple names”.

2.7.1. Resolution of simple names

Names used in ADL definitions for the declaration of sub-components and/or the spoecification of interface signatures have to be resolved into fully-qualified-names. The way mindc toolchain proceed to this name resolution is as follow :

1. If the name contains at least one dot (`"."`), then it is a fully-qualified-name. Therefore, there is nothing to do.
2. Otherwise, the name is a simple-name. In this case, for each import statement found in the ADL file (respecting the order of declaration):
   a. If the import statement contains a fully-qualified-name whose simple-name matches the name to be resolved, then the resolved fully-qualified-name is the one imported by this statement.
   b. If the import statement contains a package import (a package-name followed by `".*"`) and there exist an ADL or IDL definition whose fully-qualified-name is made of the imported package-name and the referenced simple-name, then the resolved fully-qualified-name is this one.
   c. Otherwise, try the next import.
3. If the simple name has not been resolved by the import statements, try to find an ADL or an IDL in the same package as the current definition.
4. Finally, try to find an ADL or IDL in the default package (the package with an empty name)

2.8. Definition inheritance

The ADL has a definiton inheritence feature in order to support the design of reusable ADL definitions. That is, an ADL definiton can extend one or more definitions so that it inherits from the architectural elements defined by the latters. In some cases, the inherited elements may also be overriden in the inheriting definition.

The extended definitions are specified after the name of the definition using the `"extends"` keyword followed by a comma-separated list of ADL name.

The following rules apply to type, primitive and composite definitions:

- A type definition can only extend other type definitions.
- A primitive definition can only extend type and primitive definitions.
- A composite definition can only extend type and composite definitions.

When an ADL definitions extends another one, the interface, implementation, attribute, sub-component and binding declarations are all inherited. On the other hand, import statements as well as definition parameters and template variables are not inherited.

Inherited interface, sub-component and attribute declarations can be overridden by declaring new architectural element of the same kind and with the same name.

Inherrited bindings can be overridden by declaring a new binding with the same client-side.

Private-data declaration can be overridden simply by specifying a new private-data declaration.

Finnaly, source declarations are simply inherited and cannot be overridden. Nevertheless, a definition that inherits source declaration can specify additional implementation files.

Warning

When a definition overrides the private-data definition and inherits the sources of the definition it extends, it is the responsability of the developper to make sure that the new component's private-data structure is compatible with the inherited sources.

Example 2.14. Definition extension

Given the following definitions :

```
 1  type foo.bar.MyType {
 2    provides Itf1 as itf1;
 3    requires Itf2 as itf2;
 4  }
```

```
20  primitive foo.bar.APrimitive {
21    provides Itf3 as itf3;
22
23    source aPrimitiveImpl.c;
24
25    attribute int attr1 = 2;
26  }
```

```
40  primitive foo.bar.AnotherPrimitive
41      extends MyType, APrimitive {
42
43    source anotherPrimitiveImpl.c;
44
45    @Override
46    attribute int attr1 = 4;
47  }
```

The previous definition is equivalent to :

```
60  primitive foo.bar.AnotherPrimitive {
61    provides Itf1 as itf1;
62    requires Itf2 as itf2;
63    provides Itf3 as itf3;
64
65    source aPrimitiveImpl.c;
66    source anotherPrimitiveImpl.c;
67
68    attribute int attr1 = 4;
69  }
```

In the above excerpt of code:

- Line 41 specifies that the `foo.bar.AnotherPrimitive` extends the type definition `foo.bar.MyType` and the primitive definition `foo.bar.APrimitive`.
- At line 46, the declaration of the attribute `attr1` overrides the declaration of the attribute with the same name at line 25. The `@Override` annotation at line 45 specifies that the declaration of the attribute is expected to override the inherited attribute (see Section 2.12.1.1, “`@Override`” for more details.).

2.9. Definition parameters

The ADL supports definition parameters to allow component attributes to be assigned in a parameterized way. This feature improves the reusability of architecture descriptions.

The definition parameters are specified in a comma separated list between parenthesis right after the name of the definition (and before the `"extends"` keyword if any).

When a parameterized definition is referenced in another ADL definition, either for the declaration of a sub-component, or for an inheritence specification, the value of these parameters must be specified. The value specification must be done as a comma-separated list in between paranthesis either respecting the order of declaration of parameters, or in a *`name=value`* fashion.

Types of parameters are inferred from the type of the attribute they assign. A given parameter can be used several times to initialize multiple attributes if these attributes have the same type.

Example 2.15. Definition parameters

```
 1  primitive foo.bar.APrimitive(a, b) {
 2    ...
 3    attribute int attr1 = a;
 4    attribute int attr2 = b;
 5  }
```

```
10  primitive foo.bar.AnotherPrimitive(c) extends APrimitive(c, c) {
11    ...
12  }
```

```
20  composite foo.bar.AComposite(d) {
21    ...
22    contains APrimitive(a=d, b=10) as subComp1;
23    contains AnotherPrimitive(20) as subComp2;
24    ...
25  }
```

In the above excerpt of code:

- Line 1 specifies that definition `foo.bar.APrimitive` has two parameters called `a` and `b`. These parameters are used to initialize attributes `attr1` and `attr2`, respectively;
- At line 10, the definition `foo.bar.AnotherPrimitive` have one parameter called `c`. This definition extends the `foo.bar.APrimitive` and pass the `c` parameter as value of both `a` and `b` parameters;
- At line 22, parameter values are given in a *`name=value`* fashion. The sub-component `subComp1` is an instance of the `foo.bar.APrimitive` definition where the value of its `attr1` attribute is given by the `d` parameter of the composite component definition and the value of its `attr2` attribute is `10`;
- At line 23, the sub-component `subComp2` is an instance of the `foo.bar.AnotherPrimitive` definition. Values of both `attr1` and `attr2` of this component are `20`.

2.10. Generic definitions

Generic definitions is another feature of the ADL for improving the reusability of composite component definitions. Using generic definitions, programmers may define abstract composite components where only the type of some sub-components is known rather than there concrete implementations. Such generic component definitions cannot be instantiable directly. But, they can be reused in other definitions where the concrete implementations of these sub-components are specified. For instance, a generic definition may be used for describing a system architecture which contains a memory allocator component without specifying the concrete implementation details of the latter component. This generic definition can be extended later on, in another definition where a platform specific memory component is specified in order to map the defined system architecture on a given HW platform.

To turn a composite component definition into a generic definition, a list of *template variables* should be defined between inferior and superior symbols (`"<"`, `">"`) right after the name of the component and before the `"extends"` keyword or the definition parameters, if any. Each template variable must specify the type definition which it *conforms* to.

Example 2.16. Template variable declaration

```
1  composite foo.bar.AComposite
2      <T conformsto Type1, U conformsto Type2> {
3    ...
4  }
```

Note

Line 2 declares two template variables called `T` and `U` that conforms to the type definitions `Type1` and `Type2` respectively.

Template variables can be used to specify the definition of a sub-component. Doing so, only the interfaces of the sub-cmponent are specifyed by the type which the template variable conforms to. The concrete definition of the sub-component will be given by the value that is assigned to the template variable when the generic definition is referenced.

When referencing a generic definitions, values of its template variables are specified between inferior and superior symbols (`"<"`, `">"`) in a comma separated list either respecting the order of declaration of template variables, or in a *`name=value`* fashion.

Values given to template variables must be a reference to a concrete ADL definition (or a reference to another template variable) that *conforms to* the type of the template variable. This means that the definition must have the same provided and required interfaces (same name, same signature and same size for collection interfaces); it may have additional provided interfaces; it may also have additional required interfaces but these latters must be optional. Note that it is neither necessary or sufficient that a definition extends (directly or transitively) the type to conform to it.

Example 2.17. Generic composite component definition

```
 1  type foo.bar.MyType {
 2    provides Itf1 as itf1;
 3    requires Itf2 as itf2;
 4  }
```

```
10  composite foo.bar.AComposite<T conformsto MyType>  {
11    provides Itf1 as itf1;
12
13    contains T as subComp1;
14    contains AnotherPrimitive as subComp2;
15
16    binds this.itf1 to subComp1.itf1;
17    binds subComp1.itf2 to subComp2.itf;
18  }
```

```
20  primitive foo.bar.APrimitive
21      extends MyType {
22    source aPrimitiveImpl.c;
23  }
```

```
30  composite foo.bar.AnotherComposite {
31    ...
32    contains AComposite<APrimitive> as aSubComp;
33    ...
34  }
```

Note

In the above excerpt of code:

- At line 10, the `foo.bar.AComposite` is a generic definition with one template variable called `T` that conforms to the type `foo.bar.MyType`;
- At line 13, the sub-component `subComp1` is an instance of the type variable `T`;
- At lines 16 and 17, bindings from or to the `subComp1` sub-component are checked with the interfaces defines in the `foo.bar.MyType` type definition. `subComp1` has a provided interface called `itf1` (this interface is defined at line 2). Therefore, the binding declared at line 16 is correct. Similarly, `subComp1` has a required interface called `itf2` (defined at line 3) so binding at line 17 is correct. Finally, the `subComp1` doesn't have any other mandatory required interfaces, so there is no missing binding in this composite component definition.
- At line 32, the sub-component `aSubComp` is an instance of the `foo.bar.AComposite` generic definition where the template parameter `T` takes `foo.bar.APrimitive` as value. This definition is valid since it conforms to the type definition `foo.bar.MyType`. Indeed it has the same provided and required interfaces and do not have additional mandatory required interface.

  Therefore, the sub-component `aSubComp` is a composite component that contains a `subComp1` sub-component that is an instance of the `foo.bar.APrimitive` definition.

The *conformsto* relationship is indeed quite restrictive. The goal of those restrictions is to ensure that whatever the value of a template variable is, the bindings defined in the generic definition are still valid (i.e. binding ends actually exist, and no binding is missing for a mandatory required interface).

Under some circumstances, it may be usefull to extend a generic definition and change the type of one of its template variables. This can be done using a `"?"` as a value of a template variable. When doing so, the definition must ensure that each occurence of the template variable is correctly overridden.

Example 2.18. Extension of generic definitions

This example reuses the definitions presented in the previous example.

```
 1  type foo.bar.MySecondType
 2      extends MyType{
 3    requires Itf3 as itf3;
 4  }
```

```
10  composite foo.bar.YetAnotherComposite<U conformsto MySecondType>
11      extends AComposite<?> {
12
13    @Override
14    contains U as subComp1;
15    ...
16
17    binds subComp1.itf3 to ...;
18  }
```

Note

In the above excerpt of code:

- At line 10, the `foo.bar.YetAnotherComposite` generic definition has one template variable called `U` that conforms to the type `foo.bar.MySecondType`;
- At line 11, the definition extends the `foo.bar.AComposite` generic definition and assignes `?` as value of the `T` template variable. This means that this template variable doesn't take a concrete value, so that each occurrence of this template variable must be overridden in the `foo.bar.YetAnotherComposite` definition;
- At line 14, the inherited specification of `subComp1` sub-component is overridden so that it becomes an instance of the `U` template variable.
- At line 17, the `itf3` required interface of `subComp1` is bound. The bindings of interfaces `itf1` and `itf2` are inherited from `foo.bar.AComposite`.

In this example, the `foo.bar.YetAnotherComposite` generic definition cannot extend `AComposite<U>` since `foo.bar.MySecondType` (the type of `U`) doesn't conform to `foo.bar.MyType` (the type of the template parameter of `AComposite`) because it has an additional mandatory required interface.

2.11. Anonymous definitions

As presented in previous sections of this document, each ADL definition has a name and must be placed in a separate file respecting a convention related to its name. This allows the mindc toolchain to find the ADL file of a component using its definition name. However, inlining the definition of a component in where an instance is defined, rather than writing it in a separate file may be handy in some circumstances. For that prupose, the ADL supports *anonymous definitions*, which is a feature inspired from the Java language. This feature is particularly useful for defining an extension specific to a sub-components instance. It may also be useful for defining simple components which have only one instance in the application architecture.

An anonymous definition can only be used for defining sub-components. In order to specify an anonymous definition for a sub-component declaration, the local-name must be followed by the specification of the definition nature (i.e either `"primitive"` or `"composite"`), itself followed by the content of the anonymous definition between curly braces (`"{"` and `"}"`).

Example 2.19. Anonymous definition

```
composite foo.bar.AComposite {
  ...
  contains as subComp1
    primitive {
      provides Itf1 as itf1;
      requires Itf2 as itf2;

      source myImplementation.c;
    }
  ...
}
```

Note

The architecture of sub-component `subComp1` is defined in an anonymous definition inlined in the surrounding definition. This annonymous definition describes a primitive component that specifies 2 interfaces and an implementation source file.

An anonymous definition can extends one other definition (multiple inheritence is not supported for anonymous definition). This is done by specifying the extended definition between the `"contains"` and the `"as"` keywords.

Example 2.20. Anonymous definition that extends another definition

```
composite foo.bar.AComposite {
  ...
  contains MyType as subComp1
    primitive {
      source myImplementation.c;
    }
  ...
}
```

Anonymous definitions can make use of the surrounding definition's parameters and template variables.

Example 2.21. Anonymous definition using definition parameters

```
composite foo.bar.AComposite(a) {
  ...
  contains primitive MyType as subComp1
    primitive {
      source myImplementation.c;
      attribute int attr1 = a;
    }
  ...
}
```

Warning

Anonymous definitions must be used sparingly since it may lead to ADL files that are difficult to read. Obviously, the definition of complex architectures is in a monolithic ADL file should be avoided for convenience and reusability reasons.

2.12. ADL Annotations

The ADL language supports *annotations* as a generic way to attach additional information on various elements of the language. The annotation system is directly inspired by Java and reuses its syntax.

An annotation declaration starts with a `"@"` symbol followed by the fully-qualified-name of the annotation. An annotation can have fields that are initialized in between parenthesis as a comma separated list of *`name`*`=`*`value`* pairs. A field value can be a literal constant (integer, boolean or string), another annotation, or an array of values.

If the annotation have a single field called `value`, then the value of this field can be directly put between parenthesis without specifying its name.

Example 2.22. Annotation

```
 1  @CFlags("-O3")
 2  composite foo.bar.APrimitive
 3      extends foo.bar.AnotherPrimitive {
 4
 5    @MyAnnotation(f1="toto", f2={12, 13, 56})
 6    provides Itf1 as itf1;
 7
 8    @Override
 9    attribute attr1 = 2;
10
11  }
```

Note

- At line 1, the `CFlags` annotation is attached to the definition. This annotation has one field that is assigned to `"-O3"`. This annotation is a predefined ADL annotation; see Section 2.12.1.3, “`@CFlags`” for more details.
- At line 5, the `MyAnnotation` annotation is attached to the specification of the `itf1` interface. This annotation has two fields `f1` and `f2`. The first field takes as value, the `"toto"` string; while the second one takes as value, an array of three integers.
- At line 8, the `Override` annotation is attached to the specification of the `attr1` attribute. This annotation has no fields. This annotation is a predefined ADL annotation; see Section 2.12.1.1, “`@Override`” for more details.

2.12.1. Predefined ADL Annotations

The mindc toolchain supports a set of predefined annotations that are listed in this section

In the following descriptions, the *Annotation fields* paragraph specifies the fields of the annotation and the *Annotation targets* paragraph specifies on which kind of element the annotation can be attached.

2.12.1.1. `@Override`

Annotation fields. No fields

Annotation targets. Attribute, Data, Sub-component, Binding

The `Override` annotation specifies that the element which it is attached to is supposed to override an inherited element. If it is not the case, an error is reported by the mindc toolchain.

Note that, this annotation is not required for an element to actually override an inherited element, it is only for verification purpose. In other words, there is no problem if a given element overrides an inherited element without declaring any `Override` annotation. Indeed, this annotation allows to be more resilient to the modifications of extended definitions. For instance let suppose that we have a definition *A* that extends a definition *B* and overrides the specification of a sub-component `subComp1` without adding the `@Override` annotation. Say the definition *B* is updated and the name of the sub-component `subComp1` is changed to `subComp2`. Then if the definition *A* is not updated accordingly, it will have two separate sub-components, namely `subComp1` and `subComp2`. On the other hand, if the `Override` annotation is declared for `subComp1` in the *A* definition, then a compilation error will occur to prevent the programmer.

2.12.1.2. `@Singleton`

Annotation fields. No fields

Annotation targets. Definition

The `Singleton` annotation specifies that the definition can be instantiated only a single time in a given application. An error is reported by the mindc toolchain if a definition with this annotation is instantiated more that once in an application.

Note

Composite definition that contains at least on singleton sub-component are implicitly declared singleton. It is recommanded to add explicitly the `Singleton` annotation on such composite definition.

Note

Instantiating a generic definition by passing a singleton definition as value result in a singleton definition.

2.12.1.3. `@CFlags`

Annotation fields. One field called `value` of type `string`

Annotation targets. Definition, Source

The `CFlags` annotation allows to specify compilation flags that must be used for the compilation of the source-files of a component definition.

This annotation can be attached to the definition, so that given flags are used for the compilation of the set of source-files of the definition. It can also be attached to an individual source specification to specify the compilation flags to be used only for a specific source file.

2.12.1.4. `@LDFlags`

Annotation fields. One field called `value` of type `string`

Annotation targets. Definition, Source

The `LDFlags` annotation allows to specify flags that must be used at the link phase of applications that contains at least one instance of this definition.

This annotation can be attached to the definition or to an individual source specification. In practice, this make no difference since the flags are used for the link of the whole application. Nevertheless, if a flag can be attached to a given source specification to indicate that this is this particular source file that requires this flag.

Specifying LD-Flags in ADL is particularly usefull for primitive components that use shared libraries in their code. It ensures that the link flag (for instance `-lm` if the code use the math library) is always present for the link of an application that use such primitive component.

2.12.1.5. `@UseIDL`

Annotation fields. One field called `value` of type `string[]`

Annotation targets. Definition

The `UseIDL` annotation allows to specify IDL interfaces that are used internally by the component implementation and that are not explicitly declared as a provided or required interface.

This annotation must be attached to a definition (usualy a definition of primitive component).

This is particularly useful if the component implementation cast pointers to interface types that are not directly accessible through the IDL of the provided or required interfaces.

2.12.1.6. `@Wrap`

Annotation fields. No fields

Annotation targets. Server interface

The `Wrap` annotation allows to specify that the implementation of a provided interface will be automatically generated wrapping each corresponding C functions into interface methods. Wrapping a function corresponds in defining a method that makes the call this function. The generated method returns the same type as the C function and generally takes the same parameters. This approach is close to the ld `--wrap` option.

Note

Wrapping a varadic function needs complementary information to know the name of the dual function. Actually, as the number of parameters of a variadic function can't be determined apriori, the call to this function can't be done directly but through a function assuring the same functionality having a `va_list` and a `format` as parameters. This specification will be done using the `@VarArgsDual` annotation (see Section 3.4.1, "`@VarArgsDual`") in the corresponding IDL.

2.13. Complete ADL grammar

The following listing describe the complete grammar of the ADL language

Top-level grammar

```
[1]       ADLFile ::= ( Import ) * Type
                    | ( Import ) * Primitive
                    | ( Import ) * Composite

[2]        Import ::= Annotations
                      'import' <ID> '.' ( <ID> '.' ) *
                      ( '*' | <ID> ) [ ';' ]
```

Type definition grammar

```
[3]          Type ::= Annotations
                      'type' FQN
                      [ 'extends' FQN ( ',' FQN ) * ]
                      TypeBody

[4]      TypeBody ::= '{' TypeElem * '}'
                    | TypeElem

[5]      TypeElem ::= ItfDef
```

Primitive definition grammar

```
[6]     Primitive ::= Annotations
                      [ 'abstract' ] 'primitive' FQN
                      [ ParamDecl ]
                      [ 'extends' PrimitiveRef ( ','
                      PrimitiveRef ) * ]
                      PrimitiveBody

[7]     ParamDecl ::= '(' <ID> ( ',' <ID>) * ')'

[8]  PrimitiveRef ::= FQN [ '(' [ ArgList ] ')' ]

[9]       ArgList ::= ArgValue ( ',' ArgValue ) *
                    | ArgAssign ( ',' ArgAssign ) *

[10]     ArgValue ::= String
                    | Integer
                    | <ID>

[11]    ArgAssign ::= <ID> '=' ArgValue

[12]PrimitiveBody ::= '{' PrimitiveElem * '}'
                    | PrimitiveElem

[13]PrimitiveElem ::= ItfDef
                    | AttrDef
                    | DataDef
                    | ImplDef
```

Composite definition grammar

```
[14]      Composite ::= Annotations
                        'composite' FQN
                        [ TmplVarDecl ]
                        [ ParamDecl ]
                        [ 'extends' CompositeRef ( ','
                        CompositeRef ) * ]
                        CompositeBody

[15]  TmplVarDecl ::= '<' TmplVarDecl ( ',' TmplVarDecl ) *
                        '>'

[16]  TmplVarDecl ::= <ID> 'conformsto' FQN

[17] CompositeRef ::= FQN
                        [ '<' [ TmplValueList ] '>' ]
                        [ '(' [ ArgList ] ')' ]

[18]TmplValueList ::= TmplValue ( ',' TmplValue ) *
                        | TmplAssign ( ',' TmplAssign ) *

[19]     TmplValue ::= CompositeRef
                        | '?'

[20]    TmplAssign ::= <ID> '=' TmplValue

[21]CompositeBody ::= '{' CompositeElem * '}'
                        | CompositeElem

[22]CompositeElem ::= ItfDef
                        | BindingDef
                        | CompDef
```

Interface grammar

```
[23]           ItfDef ::= Annotations
                        ( 'provides' | 'requires' )
                        [ 'optional' ] FQN
                        'as' <ID>
                        [ '[' [ <INTEGER_LITERAL> ] ']' ]
                        [ ';' ]
```

Attribute grammar

[24] *AttrDef* ::= *Annotations*
'attribute' *AttrType*
<ID> ['=' *AttrValue* **] [';']**

[25] *AttrType* ::= **'int' | 'string'**

[26] *AttrValue* ::= *String*
| *Integer*
| <ID>

Implementation grammar

[27] *DataDef* ::= *Annotations*
'data' (*Path* **| <INLINE_CODE>)**
[';']

[28] *ImplDef* ::= *Annotations*
'source' (*Path* **| <INLINE_CODE>)**
[';']

Binding grammar

[29] *BindingDef* ::= *Annotations*
'binds' *BindingEnd*
'to' *BindingEnd* **[';']**

[30] *BindingEnd* ::= **(<ID> | 'this') '.' <ID> ['['**
<INTEGER_LITERAL> ']']

Sub-component grammar

[31] *CompDef* ::= *Annotations*
'contains' [*CompositeRef* **]**
'as' <ID>
[*AnonymousC* **|** *AnonymousP***]**

[32] *AnonymousC* ::= *Annotations*
'{' *CompositeElem* *** '}'**

[33] *AnonymousP* ::= *Annotations*
'{' *PrimitiveElem* *** '}'**

Annotation grammar

```
[34]  Annotations ::= Annotation *

[35]   Annotation ::= '@' FQN
                      [ AnnoParams ]

[36]   AnnoParams ::= '(' [ AnnoVPairs
                      | AnnoValue ] ')'

[37]   AnnoVPairs ::= AnnoVPair ( ',' AnnoVPair ) *

[38]    AnnoVPair ::= <ID> '=' AnnoValue

[39]    AnnoValue ::= String
                      | Integer
                      | Boolean
                      | Annotation
                      | AnnoArrayV

[40]   AnnoArrayV ::= '{' [ AnnoValue ( ',' AnnoValue ) * ]
                      '}'
```

Other grammar

```
[41]          FQN ::= <ID> ( '.' <ID> ) *

[42]         Path ::= [ '.' '/' ] ( '..' '/' ) *
                      <ID> ( '/' <ID> ) * '.' <ID>

[43]       String ::= <STRING_LITERAL>

[44]      Integer ::= [ '+' | '-' ] <INTEGER_LITERAL>

[45]      Boolean ::= 'true' | 'false'
```

Chapter 3. Interface definition language

The IDL language allows to define component interface types. It is highly inspired by the C-type system. That is, most of the types that can be declared in C can be declared similarly in the IDL language (except function pointers that are not allowed by the IDL language). In addition the IDL language provides a special construct for the definition of interface since this kind of notion is not present in C.

The IDL language defines two kinds of files :

`'.itf'` file	Contains the definition of an interface. `'.itf'` files follow the same naming scheme as `'.adl'` files. That is, the definition of the interface `foo.bar.Itf1` must be located in a file called `foo/bar/Itf1.itf`. An `'.itf'` file may also contain type and constant definitions.
`'.idt'` file	Contains only type and constant definitions. `'.idt'` files are useful to define types and constants that are used by different interfaces.

3.1. Type definition

The IDL language supports the same type definition constructs as the C language, except function pointers. That is, an IDL file (either a `'.itf'` or `'.idt'` file) can define `struct`, `union`, `enum` or `typedef` using the same syntax as the C language.

3.2. Constant definition

The IDL language allows to define constant values using a `"#define"` construct which is similar to the C preprocessor directive used for macro definitions. Note that, function-like macros are not allowed.

3.3. Interface definition

Interfaces can only be defined in `'.itf'` files. Moreover, similarly to ADL files, an `'.itf'` file must contain one and only one interface definition whose *fully-qualified-name* matches the path of the file. (interface `foo.bar.Itf1` must be located in a file called `foo/bar/Itf1.itf`).

An interface is defined by the `"interface"` keyword followed by the fully-qualified-name of the interface and the list of methods inside curly braces (`"{"` and "}"). Method definitions are similar to traditional C function prototypes.

Example 3.1. Interface definition

```
1  interface foo.bar.Itf1 {
2    int meth1(int a, int b);
3    char meth2(struct s *s_ptr);
4  }
```

In the above code excerpt:

- Line 1 defines the interface `foo.bar.Itf1`.
- Line 2 defines a method called `meth1` that takes two integers as parameter and return an integer.
- Line 3 defines a method called `meth2` that takes a pointer to a `struct` as parameter and returns a character.

Every methods in a given interface must have a unique name.

3.3.1. Variadic methods

An interface may be defined using variadic methods. In this case the interface must also define the dual method (e.g. vprintf is the dual function of printf in stdio). Moreover the method must be specified using the `@VarArgsDual` annotation (see Section 3.4.1, “`@VarArgsDual`”).

Example 3.2. Variadic methods

```
interface foo.bar.Itf1 {
  @VarArgsDual(meth2)
  int meth1(char * my_sting, ...);
  char meth2(char * my_sting, va_list args);
}

```

3.3.2. Interface inheritance

An interface definition may extend another interface definition. The name of the extended interface is specified after a dash symbol (`":"`) just after the name of the interface. Methods defined in the extended interface are inherited and cannot be overridden.

Example 3.3. Interface inheritance

Given the following definitions :

```
 1  interface foo.bar.Itf1 {
 2    int meth1(int a, int b);
 3    char meth2(struct s *s_ptr);
 4  }
 5
```

```
10  interface foo.bar.Itf2 : foo.bar.Itf1 {
11    int meth3(int size, int tab[]);
12  }
13
```

The previous definition is equivalent to :

```
20  interface foo.bar.Itf2 {
21    int meth1(int a, int b);
22    char meth2(struct s *s_ptr);
23    int meth3(int size, int tab[]);
24  }
25
```

In the above code excerpt:

- Line 10 specifies that the interface `foo.bar.Itf2` extends the interface `foo.bar.Itf1`.

3.4. IDL Annotations

3.4.1. `@VarArgsDual`

Annotation fields. One field called `value` of type `string`

Annotation targets. Method

The `VarArgsDual` annotation is needed to specify the name of the dual method of a variadic method. A variadic method is a method with a variable number of arguments. Its dual method generally has only two arguments, the first being the same as the original method and the second being a list (`va_list`) gathering every other arguments.

Chapter 4. Component Programming Language

The CPL allows to capture the component notions inside traditional C source code. The CPL is a very lightweight pre-processed language on top of the C-language.

4.1. Component's private-data declaration

Component's private-data can be declared in a separated header file, or directly inline in the ADL using the `"data"` construct (see Section 2.3.1, “Private data definition” for more details).

The declaration of private-data takes the form of a declaration of a global variable called `PRIVATE` whose type is a structure.

Example 4.1. Private-data declaration

```
struct {
  int a, b;
  char c[10];
} PRIVATE;
```

4.2. Implementation of component's provided methods

Primitive components must implements every methods of its provided interface. The CPL provides a `"METH"` macro to specify that a C function correspond to the implementation of a method of a provided interface. This macro takes two parameters, the first one is the name of the provided interface (as written in the ADL file) and as second parameter, the name of the method (as written in the IDL file).

Example 4.2. Implementation of provided interfaces

Assuming that the component provides an interface called `calcItf` whose signature is defined in the following IDL :

```
interface foo.Calculator {
  int add(int a, int b);
  int sub(int a, int b);
}
```

Then the implementation of the component must implement the methods `add` and `sub` for the `calcItf`:

```
// implements method 'add' of interface 'calcItf'
int METH(calcItf, add)(int a, int b) {
  return a + b;
}

// implements method 'sub' of interface 'calcItf'
int METH(calcItf, sub)(int a, int b) {
  return a - b;
}
```

4.3. Implementation of component's private methods

A primitive component can also have internal functions that must have access to component's data or call client interfaces. Such functions are called *private methods*. A private method is declared with the `"METH"` macro taking only one parameter that correspond to the name of the private method.

Example 4.3. Private Method

```
int METH(myPrivateMeth)(int a) {
  ...
}
```

4.4. Invocation of component's methods

The invocation of component's methods is done using the `"CALL"` macro. This macro can take one or two parameters. To invoke a private method, the macro takes one parameter that correspond to the name of the private method to invoke. To invoke a method of a provided or a required interface, the macro takes two parameters where the first one must correspond to the name of the interface, and the second one must correspond the the name of the method to invoke.

Invocation of method can only be done inside an implementation of method of a provided interface or an implementation of a private method.

Example 4.4. Method invocation

```
1  int METH(calcItf, add)(int a, int b) {
2    CALL(myPrivateMeth)(a);
3    CALL(aRequiredItf, debug)("in add method");
4    return a + b;
5  }
```

In the above code excerpt:

- Line 2 invokes the private method called `myPrivateMeth` with the `a` parameter.
- Line 3 invokes the `debug` method of the required interface `aRequiredItf`.

4.5. Access to component's private-data

Component's private-data are accessed inside method implementation simply by accessing the `PRIVATE` structure.

Example 4.5. Private-data access

```
1  int METH(calcItf, add)(int a, int b) {
2    PRIVATE.a = a;
3    return a + b;
4  }
```

Line 2 assigns the private data called `a` to the value of the `a` parameter of the method.

4.6. Access to component's attributes

Component's attributes are accessed inside method implementation using the "ATTR" macro that takes as argument the name of the attribute.

Example 4.6. Attribute access

```
 1  int METH(calcItf, add)(int a, int b) {
 2    PRIVATE.a = a + ATTR(myAttribute);
 3    return a + b;
 4  }
```

Line 2 accesses the attribute called `myAttribute`.

4.7. Implementation of constructor and destructor

A primitive component can defines a constructor and/or a destructor methods that can be used to initialize and clear the private-data.

The constructor method of a primitive component is called just after the component is created. The destructor is called just before the component is destroyed.

Warning

Required interfaces may not be bound when the constructor or the destructor are called. So the implementation of these methods must not use the required interface of the component.

The constructor (reps. destructor) is declared as a function called "CONSTRUCTOR" (resp. "DESTRUCTOR") that has no parameter and that has no return type (neither `void` or anything else, similarly to object-oriented languages like C++ or Java)

Example 4.7. Implementation of constructor and destructor

```
CONSTRUCTOR() {
  int i;
  for (i = 0; i < 10; i++) {
    PRIVATE.a[i] = 0;
  }
}

DESTRUCTOR() {
  ...
}
```

4.8. Method and interface pointer

The CPL defines constructs to build method and interface pointers an invoke them.

4.8.1. Interface pointer

A Pointer to an interface of the component (either provided or required) can be retrieved using the "GET_MY_INTERFACE" macro that takes as parameter the name of the interface. The type of the value returned by the "GET_MY_INTERFACE" macro is the C-type corresponding to the signature of the interface (see ???).

An interface pointer can be invoked with the "CALL_PTR" macro that takes two parameter, the first one is the interface pointer to invoke, the second one is the name of the method of the interface.

Example 4.8. Interface pointer

```
1  int METH(calcItf, add)(int a, int b) {
2    foo_bar_Itf1 itfPointer = GET_MY_INTERFACE(myRequiredItf);
3    CALL_PTR(itfPointer, debug)("in add method");
4    return a + b;
5  }
```

In the above code excerpt:

- Line 2 get a pointer to the `myRequiredItf` and assign it to a local variable (assuming that the signature of this interface is `foo.bar.Itf1`).
- Line 3 invokes the `debug` method of the `itfPointer` interface pointer.

4.8.2. Method pointer

A pointer to a method (either a method of a provided interface or a private method) can be retrieved by using the "METH" macro (with one or two parameters) in an expression.

A variable whose type is a pointer to a method must be declared in a specific way. The variable name must be surrounded by the "METH_PTR" macro. For instance while `int (*f_ptr)(int a);` declares a pointer to a C function that takes an integer and returns an integer, the declaration of a pointer to a method with the same prototype must be written `int (* METH_PTR(f_ptr))(int a);`.

The "CALL_PTR" macro with one parameter can be used to invoke a method pointer.

Example 4.9. Method pointer

```
1  int METH(calcItf, add)(int a, int b) {
2    int (* METH_PTR(f_ptr))(int a);
3    f_ptr = METH(myPrivateMeth);
4    CALL_PTR(f_ptr)(a);
5    return a + b;
6  }
```

In the above code excerpt:

- Lline 2 declares a variable called `f_ptr` that is a pointer to a method that takes an integer and return an integer.
- Line 3, the `f_ptr` variable is initialized to point to the `myPrivateMeth` private method.
- Line 4 invokes the `f_ptr` method pointer.

Chapter 5. The mindc command.

The mindc toolchain allows to compile component-based application. That is, the toolchain is responsible to read and check ADL, IDL and CPL files, to generate the C glue-code, and to execute the C-compiler and linker. All these steps are completely automated through a single **mindc** command.

The **mindc** command has the following forms :

`mindc` [*`OPTIONS`*...] {*`adlname`* [:*`execname`*]...}

`adlname` is the name of the top-level ADL definition to compile, while *`execname`* is the name of the executable file that will be produced. If this latter is not specified, then the name of the executable file will be the simple name of the ADL and it will be located in a directory that corresponds to the package nane of the ADL. Several ADL definitions can be specified on the same command-line.

Available options are :

`[--src-path|-S]=<`*`path-list`*`>`	the search path of ADL, IDL and implementation files (see Section 5.1, “Source-path” for more details).
`[--out-path|-o]=<`*`path`*`>`	the directory where generated files will be put (default is `'.'`).
`[--target-descriptor|-t]=<`*`name`*`>`	the target descriptor (see Section 5.4, “Target-descriptor” for more details).
`--compiler-command=<`*`cmd`*`>`	the command of the C compiler (default is **gcc**).
`[--c-flags|-c]=<`*`flags`*`>`	the additional flags passed to the C compiler command. This option may be specified several times.
`--linker-command=<`*`cmd`*`>`	the command of the linker (default is **gcc**).
`[--ld-flags|-l]=<`*`flags`*`>`	the additional flags passed to the linker command. This option may be specified several times.
`[--linker-script|-T]=<`*`path`*`>`	the linker script to use (given path is resolved in source path).
`[--jobs|-j]=<`*`number`*`>`	the number of concurrent compilation jobs (default is 1).
`--check-adl`	Only check input ADL(s), do not compile.
`--def2c|-d`	Only generate source code of the given definitions.
`--def2o|-D`	Generate and compile source code of the given definitions, do not link an executable application.
`--force|-F`	Force the regeneration and the recompilation of every output files.
`--keep|-K`	Keep temporary output files in default output directory.
`--no-bin|-B`	Do not generate binary ADL/IDL ('.def', '.itfdef' and '.idtdef' files).
`-e`	Print error stack-traces (for debugging the compiler).

`--help|-h`	Print help message and exit

Example 5.1. mindc command examples

```
$ mindc --src-path=src --out-path=build foo.bar.MyApplication
```

In the above command :

- Input files (ADL, IDL an C files) will be shearched in the `src` directory.
- Output files will be placed in the `build` directory.
- The ADL of the top-level component to compile is `foo.bar.MyApplication`. That is, the definition is supposed to be located in the `src/foo/bar/MyApplication.adl` file. Moreover the produced executable will be located in the `build/foo/bar/MyApplication` file.

```
$ mindc --src-path=src --out-path=build foo.bar.MyApplication:myApp
```

The above example is similar to the previous one, except that the produced executable will be located in the `build/myApp` file.

5.1. Source-path

The source-path specifies the base directories used to determine the location of the input files (ADL, IDL or CPL). That is, the source path can be made of several directories. For example, if the toolchain must find the definition of an ADL called `foo.bar.MyComponent`, it will search for a file called `foo/bar/MyComponent.adl` in each directory of the source path

The source path is specified on the **mindc** command-line using the `--src-path` (or `-S`) option. This options must be followed by an equal sign (`'='`) itself followed by a list of paths. Paths are separated by `':'` on Unix-like systems and by `';'` on Windows systems (similarly to java classpaths). The `--src-path` (or `-S`) option can be specified several times on a given command-line.

The source path allows to easily integrate external sources like component libraries in your developments. The only thing you need is to place the external sources in a well known directory and specify it in the source-path.

5.2. The Mind Pre-Processor

Component implementation sources are written in the CPL language. This language is a lightweigh dialect on top of the C-language. In order to be compiled by a traditional C-compiler, the implementation source files must be preprocessed by a specific tool called the *Mind preprocessor*. This tools, replaces the CPL constructs like `METH`, `PRIVATE` or `CALL` by a standard C-code.

5.3. Customizing Compilation

The mindc toolchain drives the whole compilation process of the application. That is, it executes the C-compiler for each C source files to obtain object files (i.e. `.o` files) and then executes the linker to obtain an executable file.

The **mindc** command allows to customize the execution of the C-compiler and the linker using several options.

- `--compiler-command` and `--linker-command` allows to specify the command of the C-compiler and the linker (respectivly). The default value of these options is `gcc`. These options are particularly usefull to use cross-compiler.
- `--c-flags` (or `-c`) option allows to specify additional flags that must be passed to the C-compiler. This is particularly usefull to specify warning flags (`-Wall` for instance) or to define preprocessor macros (`-DFOO=2` for instance). This option can be specify several times. Note that c-flags can also be specified in ADL using the `CFlags` annotation (see Section 2.12.1.3, “`@CFlags`” for more details).
- `--ld-flags` (or `-l`) option allows to specify additional flags that must be passed to the linker. This option can be specify several times. Note that ld-flags can also be specified in ADL using the `LDFlags` annotation (see ??? for more details).
- `--linker-script` or `-T` option allows to specify the linker-script that must be passed to the linker. The linker-script file must be located in the source-path. This is allows to distribute a linker-script for several platforms in a component library alongside component sources. If you want to specify a linker-script that is not located in the source-path you can use the `--ld-flags` option instead (`--ld-flags=-TmyLinkerScript.ls` for instance).

By default, the mindc toolchain will print a short message when it execute the C-compiler or the linker. In some circumstances, it may be useful to see exactly the command-lines that are executed. This can be achieved by specifying the verbosity level. See Section 5.6.1, “Setting the verbosity level” for more details.

5.4. Target-descriptor

When developping component-based application for various embedded platforms, the managment of the compilation options presented in Section 5.3, “Customizing Compilation” can be tedious. The mindc toolchain uses the notion of *target-descriptor* to regroup all these options (and some other things) in a single file or in a hierarchy of files.

A target-descriptor file is a small XML file suffixed by `".td"`. It must be placed along-side the component sources and follows naming conventions similar to ADL files (target-descriptor `foo.bar.MyPlatform` must be located in a file called `foo/bar/MyPlatform.td`).

It allows to specify the command of the C-compiler and the linker, so as to specify various c-flags, ld-flags and linker-script. A target descriptor may also define a mapping between the input ADL and the ADL that is actually compiled by the toolchain. This is useful when an applications must be wrapped in a bootstrap component to be compiled and run correctly on a given platform.

A target descriptor may extends one or more other descriptors. This allows to reuse and extends existing descriptors. This is particularly useful when many different target platforms partially shares the same hardware. This can also be usefull to defines different compilation profiles for a given target platform. For instance: a debug profile with debug information in binary file and without optimization; and a production profile without debug info and with optimizations.

5.4.1. Target descriptor syntax

A target descriptor must start with the following XML prolog :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE target SYSTEM
  "classpath://org/ow2/mind/target/ast/targetDescriptor.dtd">
```

Note

The DTD URL must be written on a single line without linebreak.

A target descriptor has a very simple XML structure. The top-level element is called `"target"` It must have a `"name"` attribute that specify its name.

For example:

```
<target name="foo.bar.MyPlatform">
  ...
</target>
```

The `"target"` top-level elements can contains the following sub-elements (in this order):

`extends`	Specify an extended target descriptor. A target-descriptor may contains several `extends` sub-elements. Attributes: `name` the name of the extended target descriptor.
`adlMapping`	Defines a mapping from the *input ADL* to the ADL that is actually compiled. Attributes: `mapping` the ADL mapping. Use `$inputADL` to refer to the input ADL.
`compiler`	Specifies the C-compiler command Attributes: `path` the C-compiler command
`linker`	Specifies the linker command Attributes: `path` the linker command
`linkerScript`	Specifies the linker script Attributes: `path` the linker script
`cFlag`	Specifies a set of c-flags. A target-descriptor may contains several `cFlag` sub-elements. Attributes: `id` the identifier of this set of c-flags (optional) `value` the c-flags
`ldFlag`	Specifies a set of ld-flags. A target-descriptor may contains several `ldFlag` sub-elements. Attributes: `id` the identifier of this set of ld-flags (optional) `value` the ld-flags

Example 5.2. Simple target-descriptor

```
1 <target name="unix">
2   <adlMapping
3     mapping="unix.boot.BootstrappedApplication<${inputADL}>" />
4   <compiler path="gcc" />
5   <linker path="gcc" />
6   <cFlag id="debug" value="-g" />
7   <cFlag id="warning" value="-Wall" />
8 </target>
```

In the above target-descriptor

- Lines 2 and 3 define a mapping from the input ADL (the ADL given on the **mindc** command line), and the ADL that is actually compiled by the toolchain. So if the command line is

  ```
  mindc --target-descriptor=unix foo.bar.MyComponent
  ```

  Then the ADL that is actually compiled by the toolchain will be `unix.boot.BootstrappedApplication<foo.bar.MyComponent>`

- Lines 4 and 5 define `gcc` as the C-compiler and linker command.

- Line 6 defines a set of c-flags with the `debug` identifier and with `"-g"` as value. The `debug` identifier is used to give a name to this set of c-flags. This allows other target-descriptors to extend this one and override this set of c-flag. See Section 5.4.2, “Target-description extension” for more details.

5.4.2. Target-description extension

A target-descriptor can extends one or more other target-descriptor(s) using the `extends` sub-element. Elements of extended target-descriptors are inherited and can be overridden. `adlMappind`, `compiler`, `linker` and `linkerScript` elements can be overridden simply by specifying a new element of the same kind.

`cFlag` and `ldFlag` are handle differently. Only `cFlag` (or `ldFlag`) elements that have an `id` attribute can be overridden by specifying a new `cFlag` (or a new `ldFlag` respectively) with the same `id` attribute. `cFlag` and `ldFlag` that do not have an `id` cannot be overridden and are simply inherited.

Example 5.3. Target-descriptor extension

```
1 <target name="unix.warning">
2   <extends name="unix"/>
3   <cFlag id="warning" value="-Werror -Wall -Wredundant-decls" />
4 </target>
```

In the above target descriptor

- Line 2 specifies that this target-descriptor extends the target-descriptor called `unix` (presented in Example 5.2, “Simple target-descriptor”).

- Line 3 overrides the specification of the c-flags with the `id` `"warning"` (see line 7 in Example 5.2, “Simple target-descriptor”). This allows to specify more strict warning flags.

So using the `unix.warning` target descriptor instead of the `unix` one, the toolchain will use the same compilers and linker, the same ADL mapping, the same c-flags for debug (i.e. the `-g` flag at line 6 in Example 5.2, “Simple target-descriptor”), but will use more strict warning flags.

5.5. Generated files

The mindc toolchain generates many files (the generated executable file but also many C source files and compilation files). These files are located in the *output directory* that is specified on the command-line with the `--out-path` option. This section gives a brief description of these generated files.

5.5.1. Generated source files

When the mindc toolchain compiles an ADL, it will generate various C source files for each ADL that is used directly or indirectly by the top-level ADL. The name of these generated files follows the same name-scheme as the ADL. That is, every files generated for an ADL called `foo.bar.MyComponent`, will be called `foo/bar/MyComponent_<suffix>`

For each ADL, the toolchain generates the following files:

`<ADL prefix>.adl.h`	The C header file that contains the definition of the component structure. If the ADL correspond to a primitive component that have private-data, then this file includes the file that contains the definition of the private-data.
`<ADL prefix>.inc`	The C header file that includes the `.adl.h` file and that declares every methods of the provided interface of the component. This file in automatically included by the implementation files of the component.
`<ADL prefix>_ctrl_impl.c`	The C source file that contains the implementation code of the membrane of the component (code for controllers and factories).
`<ADL prefix>.macro`	The C header file that contains the definition of the CPP macros that are used in the code generated by the Mind Pre-Processor.

Moreover, for the top-level ADL, the toolchain generates additional files in which every component instances are correctly initialized. For each ADL used by the top-level ADL (directly or indirectly), the toolchain will generate the following file.

`<top-level ADL>_<used ADL>_instances.c`	This file contains the initialization of the component instances of the `<used ADL>` ADL in the application `<top-level ADL>`.

5.5.2. Generated compilation files

When compiling generated source files and user implementations of components, the toolchain generates various intermediate files, in addition to the traditional object file (`.o` file). This intermediate files are used or generated by the Mind-preprocessor.

Compilation files of user implementation of components (i.e. source files referenced in ADL using the `source` are called `<ADL name>_impl<NUM>.<ext>`, where .`<NUM>` is the index of the source file in the ADL (since implementation of primitive components may be splitted in several files). While compilation files of generated source files have the same name as the source file (except the extension).

The following compilation files are generated:

`<filename>.i`	The C-preprocessed file. This file contains the C-source afer the execution of the C-preprocessor. That is, every C-preprocessor directives (`#include`, `#define`, ...) are replaced. This file is the input file of the Mind-preprocessor.

`<filename>.mpp.c`	The Mind-preprocessed file. This file contains the C-source afer the execution of the Mind-preprocessor, where every CPL macros (`METH`, `CALL`, ...) are replaced. This file is the C-source file that is actually compiled by the C-compiler.
`<filename>.o`	The object file. This file is the file generated by the C-compiler.

5.5.3. Generated files for generic ADLs

The prefix of the generated files for generic ADLs is more complex. Indeed, since a generic ADL definition may by used in different contexts, with different values for its template variables, the generated source-code must be different for each set of values of its template variables. For instance, source-code generated for `MyGeneric<ADL1, ADL2>` must be difiirent from the code generated for `MyGeneric<ADL3, ADL4>`.

That is, the name of generated files for generic ADLS will match `<generic ADL name>_tmpl_<hashcode>` where `<hashcode>` is different for each set of values of its template variables. This hashcode is based on the values of the template variables (i.e. the hashcode of `<ADL1, ADL2>` or `<ADL3, ADL4>` in previous examples).

If a given generic ADL definition is used several times with different value for its template variables, then it may be difficult to figure out which files correspond to which component. To solve this problem, the toolchain will generate a file that is called `<generic ADL name>.map` that details the value of the template variables that correspond to a given hascode.

5.6. Environment variables

Two environment variables are available to adapt the behavior of the mindc toolchain:

`MIND_CLASSPATH`	Allows to add Java classes to the classpath used to execute the toolchain. This can be used to extends the toolchain.
`MIND_OPTS`	Allows to pass additional parameters to the Java virtual machine that runs the toolchain. This can be used, in particular to specify the verbosity level of the toolchain (see Section 5.6.1, “Setting the verbosity level” for more details).

5.6.1. Setting the verbosity level

By default, the mindc toolchain print very few messages. It only print a simple line for each compilation tasks. Nevertheless, the toolchain uses internal a highly configurable logging system that can be customized using the `MIND_OPTS` environment variable.

By setting the `MIND_OPTS` environment variable to the following value, the toolchain will print more detailled messages. In particular it will print the actual command line that is executed for each compilation task.

```
-Ddefault.console.level=FINE
```

Chapter 6. Advanced Features

An ADL definition describes a static architecture where every components are instantiated statically in the compiled binary. The Fractal component model defines components as runtime entities. That is, components of an application are entities that can be manipulated during its execution. In particular, it is possible to create and destroy component dynamically using *component factory* and it is also possible to introspect and reconfigure components using the *Fractal controllers*.

6.1. Component Factory

A component factory is a component that provides an interface that can be used to instantiate or destroy new components. A component factory is defined using a template-like notation: `Factory<`*`instantiated ADL`*`>`. A component factory provides an interface called `factory` whose signature is defined in the IDL called `fractal.api.Factory`. A component factory also requires an interface called `allocator` whose signature is defined in the IDL called `memory.api.Allocator`. This required interface is by the component factory to allocate/free memory to store components data.

See Section 7.2.6, “The `fractal.api.Factory` interface” for details on the `fractal.api.Factory` interface signature.

Example 6.1. Declaration of Component Factory in ADL

```
 1  composite foo.bar.MyComposite {
 2    ...
 3    contains Factory<AComponent> as myComponentFactory;
 4    contains ... as myFactoryUser;
 5    contains ... as myAllocator;
 6
 7    binds myFactoryUser.factory to myComponentFactory.factory;
 8    binds myComponentFactory.allocator to myAllocator.allocator;
 9    ...
10
11  }
```

In the above code excerpt:

- Line 3 declares a sub-component called `myComponentFactory` which is a *component factory* that instantiate components describes in the ADL called `AComponent`.
- Line 7 declares a binding from the `myFactoryUser` component to the `factory` interface of the `myComponentFactory`. This allows the `myFactoryUser` component to use the factory component to create/destroy components that are described in the ADL called `AComponent`.
- Line 8 declares a binding from the `allocator` provided interface of the `myComponentFactory` component to an `allocator` interface provided by the `myAllocator` component.

The implementation of the `myFactoryUser` component may look-like :

Example 6.2. Use of Component Factory

```
 1  int err;
 2  void *instance;
 3  err = CALL(factory, newFcInstance)(&instance);
 4  if (err != 0) {
 5    // an error occurs
 6    return err;
 7  }
 8
 9  // do something with this new instance
10  // ...
11
12  err = CALL(factory, destroyFcInstance)(instance);
13  if (err != 0) {
14    // an error occurs
15    return err;
16  }
```

In the above code excerpt:

- Line 3 calls the `newFcInstance` method of the component factory to create a new component and store a pointer to it in the `instance` variable.
- Line 12 calls the `destroyFcInstance` method of the component factory to destroy the component that is pointed by the `instance` variable.

By default a component factory as no controller interface, it has only the `factory` provided interface and the `allocator` required interface. In some circumstance, it can be useful to have component factory that have some controller interfaces that allows to introspect and reconfigure it dynamically. This can be done by using `FactoryWithCtrl` instead of `Factory`. For instance :

```
contains FactoryWithCtrl<AComponent> as myComponentFactory;
```

6.2. Fractal Controllers

A controller is an additional interface provided by a component that allows to introspect or reconfigure it. The Fractal component model defines a set of standard controller interfaces.

The mindc toolchain allows to specify in the ADL, the controllers provided by components using annotations. Controllers can be specified individualy using one annotation per controller, or can be specified all-together using a single annotation (see Section 6.2.6, “The standard Fractal controllers”).

6.2.1. The `component` controller

The `component` controller allows to introspect the interfaces provided by the component. To add a `component` controller on a component, the `@controller.Component` annotation must be attached to the ADL definition.

Example 6.3. Declaration of the `component` controller in ADL

```
1  @controller.Component
2  primitive foo.bar.MyControlledComponent {
3    ...
4  }
```

Line 1 specifies that a `component` controller must be added to the component `foo.bar.MyControlledComponent`.

When the `@controller.Component` annotation is attached to an ADL definition, a control interface called `component` is automatically added. The signature of this interface is defined in the `fractal.api.Component` IDL.

See Section 7.2.1, “The `fractal.api.Component` interface” for details on the `fractal.api.Component` interface signature.

When the `@controller.Component` annotation is attached to an ADL definition, the toolset ensures that the added `component` interface as the first interface of the component. This allows to cast a pointer to a component (as returned by a component factory) to a pointer to a `fractal.api.Component` interface.

Example 6.4. Use the `component` interface of a component created by a factory component

```
composite foo.bar.MyComposite {
  ...
  contains Factory<foo.bar.MyControlledComponent>
      as myComponentFactory;
  contains ... as myFactoryUser;
  contains ... as myAllocator;

  binds myFactoryUser.factory to myComponentFactory.factory;
  binds myComponentFactory.allocator to myAllocator.allocator;
  ...

}
```

```
 1 int err;
 2 void *instance;
 3 fractal_api_Component componentItf;
 4 int nbItfs;
 5 err = CALL(factory, newFcInstance)(&instance);
 6 if (err != 0) {
 7   // an error occurs
 8   return err;
 9 }
10
11 componentItf = (fractal_api_Component) instance;
12 nbItfs = CALL_PTR(componentItf, listFcInterfaces)(NULL);
13 if (nbItfs <) {
14   // an error occurs
15   return nbItfs;
16 }
17 ...
```

In the above C-code excerpt:

- Line 3 declares a variable called `componentItf` that is a pointer to a `fractal.api.Component` interface.
- Line 11 casts the `instance` pointer to a pointer to a `fractal.api.Component` interface. This cast works correctly since components instantiated by the factory have a `component` controller and this controller guaranties that the `component` control interface is the first interface of the component.
- Line 12 calls the `listFcInterfaces` method of the `component` control interface of the component created at line 5.

6.2.2. The `binding-controller`

The `binding-controller` allows to introspect and reconfigure the bindings of the required interfaces of the component. To add a `binding-controller` on a component, the `@controller.BindingController` annotation must be attached to the ADL definition.

Example 6.5. Declaration of the `binding-controller` in ADL

```
1  @controller.BindingController
2  primitive foo.bar.MyControlledComponent {
3    ...
4  }
```

Line 1 specifies that a `binding-controller` must be added to the component `foo.bar.MyControlledComponent`.

When the `@controller.BindingController` annotation is attached to an ADL definition, a control interface called `bindingController` is automatically added. The signature of this interface is defined in the `fractal.api.BindingController` IDL.

See Section 7.2.2, “The `fractal.api.BindingController` interface” for details on the `fractal.api.BindingController` interface signature.

By default, the `@controller.BindingController` annotation can be specified only on components that have at least one required interface. The annotation has an optional field called `allowNoRequiredItf` that can be set to `true` to bypass this limitation (i.e. `@controller.BindingController(allowNoRequiredItf=true)`).

6.2.3. The `content-controller`

The `content-controller` allows to introspect and reconfigure the sub-components and internal bindings of a composite component. To add a `content-controller` on a component, the `@controller.ContentController` annotation must be attached to the ADL definition.

Example 6.6. Declaration of the `content-controller` in ADL

```
1  @controller.ContentController
2  primitive foo.bar.MyControlledComponent {
3    ...
4  }
```

Line 1 specifies that a `content-controller` must be added to the component `foo.bar.MyControlledComponent`.

When the `@controller.ContentContrller` annotation is attached to an ADL definition, a control interface called `contentController` is automatically added. The signature of this interface is defined in the `fractal.api.ContentController` IDL.

See Section 7.2.3, “The `fractal.api.ContentController` interface” for details on the `fractal.api.ContentController` interface signature.

The `@controller.ContentController` annotation imposes various limitations on the content of the composite component to which it is attached. Indeed, each sub-component of the composite must have a `component` controller and if it has at least one client interface, a `binding-controller`.

The current implementation of the `content-controller` can manage only a fixed number on sub-components. This number is controlled by the `nbDynamicSubComponent` field of the `@controller.ContentController` annotation. This annotation field control the number of additional sub components that can be managed by the composite. So at runtime, the composite may contain at most `nbDynamicSubComponent + nbInitialSubComponent` sub-components (where `nbInitialSubComponent` is the number of sub-component described in the ADL of the composite). The default value of this field is 10.

Moreover, the current implementation of the `removeFcSubComponent` method does not check that the removed sub-component is no more bound to another sub component inside the composite.

6.2.4. The `life-cycle-controller`

The `life-cycle-controller` allows to introspect and change the state (started/stopped) of the component. To add a `life-cycle-controller` on a component, the `@controller.Component` annotation must be attached to the ADL definition.

Example 6.7. Declaration of the `life-cycle-controller` in ADL

```
1  @controller.LifeCycleController
2  primitive foo.bar.MyControlledComponent {
3    ...
4  }
```

Line 1 specifies that a `life-cycle-controller` must be added to the component `foo.bar.MyControlledComponent`.

When the `@controller.LifeCycleController` annotation is attached to an ADL definition, a control interface called `lifeCycleController` is automatically added. The signature of this interface is defined in the `fractal.api.LifeCycleController` IDL.

See Section 7.2.5, "The `fractal.api.LifeCycleController` interface" for details on the `fractal.api.LifeCycleController` interface signature.

When the `@controller.LifeCycleController` annotation is attached to a definitions of primitive compoent, the source code must contain at least the implementation of one of the `startFc` or `stopFc` methods of the `lifeCycleController` interface. For instance:

Example 6.8. Implementation of the `life-cycle-controller` in primitive component.

```
int METH(lifeCycleController, startFc) (void) {
  printf("in startFc\n");
  // ...
  return FRACTAL_API_OK;
}

int METH(lifeCycleController, stopFc) (void) {
  printf("in stopFc\n");
  // ...
  return FRACTAL_API_OK;
}
```

If a composite contains at least one sub-component that provides a life-cycle controller, then a life-cycle controller is automatically added on the composite definition. So in most cases, it is not necessary to declare explicitly the life-cycle controller on composite definition since it would be added automatically when necessary.

6.2.5. The `attribute-controller`

The `attribute-controller` allows to introspect and modify the attributes of a component. To add an `attribute-controller` on a component, the `@controller.AttributeController` annotation must be attached to the ADL definition.

Example 6.9. Declaration of the `attribute-controller` in ADL

```
1  @controller.AttributeController
2  primitive foo.bar.MyControlledComponent {
3    ...
4  }
```

Line 1 specifies that a `attribute-controller` must be added to the component `foo.bar.MyControlledComponent`.

When the `@controller.AttributeController` annotation is attached to an ADL definition, a control interface called `attributeController` is automatically added. The signature of this interface is defined in the `fractal.api.AttributeController` IDL.

See Section 7.2.4, “The `fractal.api.AttributeController` interface” for details on the `fractal.api.AttributeController` interface signature.

By default, the `@controller.AttributeController` annotation can be specified only on components that have at least one attribute. The annotation has an optional field called `allowNoAttr` that can be set to `true` to bypass this limitation (i.e. `@controller.AttributeController(allowNoAttr=true)`).

6.2.6. The standard Fractal controllers

Specifying individual controllers on each component definition may be laborious. The `@controller.StdControllers` annotation can be used to specify that a component must provide every standard Fractal controllers according to its architecture.

When the `@controller.StdControllers` annotation is attached to an ADL definition, the following controllers will be added to the component:

- the `component` controller (see Section 6.2.1, “The `component` controller”)
- the `binding-controller` controller, if the component has at least one required interface, (see Section 6.2.2, “The `binding-controller`”)
- the `content-controller` controller, if the component is a composite, (see Section 6.2.3, “The `content-controller`”)
- the `attribute-controller` controller, if the component has at least one attribute, (see Section 6.2.5, “The `attribute-controller`”)

The `life-cycle-controller` is never added automatically by the `@controller.StdControllers`. Indeed, the `life-cycle-controller` must be specified explicitly on primitive component since it requires that the implementation code contains at least the implementation of one of the `startFc` or `stopFc` methods.

The `@controller.StdControllers` annotation is inherited, this means that the added controllers depends on the architecture of the final definition in the inheritence graph and not the definition (in this inheritence graph) on which the annotation is added. For example:

Example 6.10. `@controller.StdControllers` and definition inheritence

```
@controller.StdControllers
type foo.bar.MyType {
  requires foo.Itf1 as aRequiredInterface;
}
```

```
primitive foo.bar.MyPrimitive extends foo.bar.MyType {
  ...
  attribute int myAttr;
}
```

```
composite foo.bar.MyComposite extends foo.bar.MyType {
  ...
}
```

In the above ADLs:

- The `foo.bar.MyPrimitive` component will have the following controllers : `component`, `binding-controller` and `attribute-controller`
- The `foo.bar.MyComposite` component will have the following controllers : `component`, `binding-controller` and `content-controller`
- Moreover, if the `foo.bar.MyType` type is used as the type of a template variable, then this template variable provides the `component` and `bindingController` interfaces. For instance :

```
composite foo.bar.MyTemplate<T conformsto foo.bar.MyType> {
  ...
  contains T as subComp;
  ...
  binds ... to subComp.component;
  binds ... to subComp.bindingController;
}
```

Chapter 7. Fractal runtime reference

7.1. The default package

7.1.1. The `mindcommon.h` file

This file contains MIND common types and macro definitions. It is included by generated code. These definitions can be configured thank to various "configuration macros". These configuration macros can be set in various way :

- from the mindc command-line with the --c-flag option (see Section 5.3, “Customizing Compilation”);
- in a target descriptor to specialize these common definitions for a given platform or C compiler (see Section 5.4, “Target-descriptor”);
- in an ADL definition with the `@CFlags` annotation do specialize these common definitions for a given component (see Section 2.12.1.3, “`@CFlags`”).

This file contains the following code :

```
#ifdef __MIND_USERINCLUDE_H
#include __MIND_USERINCLUDE_H
#endif
```

If the `__MIND_USERINCLUDE_H` macro is defined, it must be a path to an header file that is included here. This header file may contain macro definitions used in the rest of this file.

```
#ifndef __MIND_NO_STDINT_H
#ifndef __MIND_STDINT_H
#define __MIND_STDINT_H <stdint.h>
#endif
#include __MIND_STDINT_H
#endif
```

Include `<stdint.h>`.

The `__MIND_NO_STDINT_H` macro can be defined to avoid the inclusion of `<stdint.h>`.

The `__MIND_STDINT_H` macro can be defined to override the path to the `<stdint.h>` file.

```
#ifndef __MIND_NO_STDDEF_H
#ifndef __MIND_STDDEF_H
#define __MIND_STDDEF_H <stddef.h>
#endif
#include __MIND_STDDEF_H
#endif
```

Include `<stddef.h>`.

The `__MIND_NO_STDDEF_H` macro can be defined to avoid the inclusion of `<stddef.h>`.

The `__MIND_STDDEF_H` macro can be defined to override the path to the `<stddef.h>` file.

```
#ifndef __MIND_STRING_TYPE
#define __MIND_STRING_TYPE const char *
```

```
#endif
#ifndef __MIND_NO_STRING_TYPEDEF
typedef __MIND_STRING_TYPE string;
#define __MIND_STRING_TYPEDEF string
#define __MIND_CONST_STRING_TYPEDEF const string
#else
#define __MIND_STRING_TYPEDEF __MIND_STRING_TYPE
#define __MIND_CONST_STRING_TYPEDEF __MIND_STRING_TYPE const
#endif
```

Defines the `string` type.

The `__MIND_STRING_TYPE` macro can be defined to override the default definition of the `string` type. (default definition is `const char *`).

The `__MIND_NO_STRING_TYPEDEF` macro can be defined to avoid the definition of the `string` typedef.

Note that the code generated by mindc, do not use the `string` typedef directly. It uses instead the `__MIND_STRING_TYPEDEF` macro. This macro expands to `__MIND_STRING_TYPE` if the `__MIND_NO_STRING_TYPEDEF` macro is defined, otherwise it expands to `string`.

```
#ifdef __MIND_STRICT_C89
#define __MIND_NO_INLINE
#define __MIND_NO_GCC_ATTRIBUTE
#endif
```

The `__MIND_STRICT_C89` macro can be used to force the generated code to be strictly compliant with C89. If this macro is defined, this implies that the `__MIND_NO_INLINE` and the `__MIND_NO_GCC_ATTRIBUTE` macros are defined.

```
#ifndef __MIND_INLINE
#ifdef __MIND_NO_INLINE
#define __MIND_INLINE
#else
#define __MIND_INLINE inline
#endif
#endif
```

Defines the `__MIND_INLINE` macro. This macro expands to the `inline` C keyword, and is used in the code generated by mindc.

The `__MIND_NO_INLINE` macro can be defined to avoid the use of `inline` keyword in the generated code. (This is required to compile code in stdc89)

```
#ifndef __MIND_ATTRIBUTE
#ifdef __MIND_NO_GCC_ATTRIBUTE
#define __MIND_ATTRIBUTE(attr)
#else
#define __MIND_ATTRIBUTE(attr) __attribute__(attr)
#endif
#endif
```

The `__MIND_NO_GCC_ATTRIBUTE` macro can be defined to avoid the use of gcc attribute in the generated code. Note that if the `__MIND_NO_GCC_ATTRIBUTE` is defined, the compilation of the generated code may raise some warnings when compiled with GCC with the "-Wall" flag.

```
#ifndef __MIND_ATTRIBUTE_UNUSED
#define __MIND_ATTRIBUTE_UNUSED __MIND_ATTRIBUTE((unused))
#endif
```

The `__MIND_ATTRIBUTE_UNUSED` macro is used to add the `unused` GCC attribute to some generated functions.

7.2. The `fractal.api` package

7.2.1. The `fractal.api.Component` interface

This interface use constants defined in `fractal/api/ErrorConst.idt`.

This interface defines the following methods:

```
int getFcInterface(in const string interfaceName,
  out void* interfaceReference);
```

The `getFcInterface` method returns an external interface of the component to which this interface belongs.

`interfaceName`	the name of the external interface that must be returned.
`interfaceReference`	(out parameter) the external interface of the component to which this interface belongs, whose name is equal to the given name.
return	`0` if the interface exist and has been returned correctly or `FRACTAL_API_NO_SUCH_INTERFACE` if there is no such interface.

```
int listFcInterfaces(in const string interfaceNames[]);
```

The `listFcInterfaces` method returns the names of the external interfaces of the component to which this interface belongs. More precisely, if the given `interfaceNames` is `null`, this method returns the number of external interfaces. If it is not `null`, this method assumes that the given array is big enough to contain all the names of the external interfaces.

`interfaceNames`	an array into which names of the external interfaces are copied. Can be `null`.
return	the number of interface names; or `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int getFcInterfaceRole(in const string interfaceName);
```

The `getFcInterfaceRole` method returns the role of an external interface of the component to which this interface belongs.

`interfaceName`	the name of an external interface.
return	`0` if the interface exist and is a client interface; `1` if the interface exist and is a server interface; or `FRACTAL_API_NO_SUCH_INTERFACE` if there is no such interface.

```
int getFcInterfaceSignature(in const string interfaceName,
  out const string signature);
```

The `getFcInterfaceSignature` method returns the signature of an external interface of the component to which this interface belongs.

`interfaceName`	the name of an external interface.
`signature`	(out parameter) the signature of the external interface of the component to which this interface belongs, whose name is equal to the given name.the signature of the external interface.
return	`0` if the interface exist and its signature has been returned correctly or `FRACTAL_API_NO_SUCH_INTERFACE` if there is no such interface.

```
int getFcInterfaces(in void* interfaceReferences[]);
```

The `getFcInterfaces` method returns the external interfaces of the component to which this interface belongs. More precisely, if the given `interfaceReferences` is `null`, this method returns the number of external interfaces. If it is not `null`, this method assumes that the given array is big enough to contain all the external interfaces.

`interfaceReferences`	an array into which references of the external interfaces are copied. Can be `null`.
return	the number of interface references; or `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

7.2.2. The `fractal.api.BindingController` interface

This interface use constants defined in `fractal/api/ErrorConst.idt`.

This interface defines the following methods:

```
int listFc(in const string clientItfNames[]);
```

The `listFc` method returns the names of the client interfaces of the component to which this interface belongs. More precisely, if the given `clientItfNames` is `null`, this method returns the number of client interfaces. If it is not `null`, this method assumes that the given array is big enough to contain all the client interface names.

`clientItfNames`	an array into which client interface names are copied. Can be `null`.
return	the number of client interfaces; or `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int lookupFc(in const string clientItfName,
  out void* interfaceReference);
```

The `lookupFc` method returns the interface to which the given client interface is bound. More precisely, returns the server interface to which the client interface whose name is given is bound. This server interface is necessarily in the same address space as the client interface.

`clientItfName`	the name of a client interface of the component to which this interface belongs.
`interfaceReference`	(out parameter) the server interface to which the given interface is bound.
return	`0` if the operation succeed. `FRACTAL_API_NO_SUCH_INTERFACE` if there is no such client interface. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int bindFc(in const string clientItfName, in void* serverItf);
```

The `bindFc` method binds the client interface whose name is given to a server interface. More precisely, binds the client interface of the component to which this interface belongs, and whose name is equal to the given name, to the given server interface. The given server interface must be in the same address space as the client interface.

`clientItfName`	the name of a client interface of the component to which this interface belongs.
`serverItf`	a server interface.
return	`0` if the operation succeed. `FRACTAL_API_NO_SUCH_INTERFACE` if there is no such client interface. `FRACTAL_API_ILLEGAL_BINDING` if the binding cannot be created. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if this component has a {`LifeCycleController` interface, but it is not in an appropriatestate to perform this operation.

```
int unbindFc(in const string clientItfName);
```

The `unbindFc` method unbinds the given client interface. More precisely, unbinds the client interface of the component to which this interface belongs, and whose name is equal to the given name.

`clientItfName`	the name of a client interface of the component to which this interface belongs.
return	`0` if the operation succeed. `FRACTAL_API_NO_SUCH_INTERFACE` if there is no such client interface. `FRACTAL_API_ILLEGAL_BINDING` if the binding cannot be created. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if this component has a {`LifeCycleController` interface, but it is not in an appropriatestate to perform this operation.

7.2.3. The `fractal.api.ContentController` interface

This interface use constants defined in `fractal/api/ErrorConst.idt`.

A component interface to control the content of the component to which it belongs. This content is supposed to be made of an unordered, unstructured set of components and bindings.

This interface defines the following methods:

```
int getFcSubComponents(in fractal.api.Component subComponents[]);
```

The `getFcSubComponents` method returns the sub-components of the component to which this interface belongs. More precisely, if the given `subComponents` is `null`, this method returns the number of sub components. If it is not `null`, this method assumes that the given array is big enough to contain all the references to the `Component` interfaces of the sub components.

`subComponents`	an array into which the references to the `fractal.api.Component` interfaces of the sub components are copied. Can be `null`.
return	the number of sub components; or `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int getFcSubComponent(in string name, out fractal.api.Component subComponent);
```

The `getFcSubComponent` method returns the sub-component of the component to which this interface belongs and which has the given `name`.

`name`	the name of the sub-component to return.
`subComponent`	(out parameter) the `fractal.api.Component` interface of the sub-component.
return	`0` if the sub-component has been returned correctly. `FRACTAL_API_NO_SUCH_SUB_COMPONENT` if no sub-component with the given name can be found. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int getFcSubComponentName(in fractal.api.Component subComponent,
  out string name);
```

The `getFcSubComponentName` method returns the name of the given sub-component.

`subComponent`	the `fractal.api.Component` interface of a sub-component.
`name`	(out parameter) the name of the given sub-component.
return	`0` if the name of the sub-component has been returned correctly. `FRACTAL_API_NO_SUCH_SUB_COMPONENT` if the given `subComponent` is not a sub component. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int addFcSubComponent(in fractal.api.Component subComponent);
```

The `addFcSubComponent` method adds a sub-component to this component. More precisely adds the component whose reference is given as a sub-component of the component to which this interface belongs. If `C` is the sub-component set returned by `getFcSubComponents` just before a call to this method, and `C'` is the sub-component set just after this call, then `subComponent` is guaranteed to be in `C'`, but `C'` is *not* guaranteed to be the union of `C` and `{subComponent}`, nor to contain all the elements of `C`.

`subComponent`	the component to be added inside this component.
return	`0` if the component is added correctly. `FRACTAL_API_ILLEGAL_CONTENT` if the component cannot be added. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if this component has a interface, but it is not in an appropriate state to perform this operation. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int addFcSubComponent(in fractal.api.Component subComponent);
```

The `addFcSubComponent` method adds a sub-component to this component with the given local-name. More precisely adds the component whose reference is given as a sub-component of the component to which this interface belongs. If `C` is the sub-component set returned by `getFcSubComponents` just before a call to this method, and `C'` is the sub-component set just after this call, then `subComponent` is guaranteed to be in `C'`, but `C'` is *not* guaranteed to be the union of `C` and `{subComponent}`, nor to contain all the elements of `C`.

`subComponent`	the component to be added inside this component.
`name`	the local-name of the added sub-component. May be `null`.
return	`0` if the component is added correctly. `FRACTAL_API_ILLEGAL_CONTENT` if the component cannot be added. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if this component has a interface, but it is not in an appropriate state to perform this operation. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int removeFcSubComponent(in fractal.api.Component subComponent);
```

The `removeFcSubComponent` method removes a sub-component from this component. More precisely removes the sub-component whose reference is given from the component to which this interface belongs. If `C` is the sub-component set returned by `getFcSubComponents` just before a call to this method, and `C''` is the sub-component set just after this call, then `subComponent` is guaranteed not to be in `C''`, but `C''` is *not* guaranteed to be the difference of `C` and `{subComponent}`, nor to contain all the elements of `C` distinct from `subComponent`.

`subComponent`	the component to be removed from this component.
`serverItf`	a server interface.
return	`0` if the component has been removed correctly. `FRACTAL_API_ILLEGAL_CONTENT` if the component cannot be removed. `FRACTAL_API_NO_SUCH_SUB_COMPONENT` if the given `subComponent` is not a sub component. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if this component has a LifeCycleController interface, but it is not in an appropriate state to perform this operation. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int addFcSubBinding(in fractal.api.Component clientComponent,
  in string clientItfName, in fractal.api.Component serverComponent,
  in string serverItfName);
```

The `addFcSubBinding` method creates a binding inside this component.

`clientComponent`	the component that contains the client interface that has to be bound. If `null` the client component is assumed to be the component this interface belongs.
`clientItfName`	the name of the interface at the client side of the binding. If the given `clientComponent` is a sub component; this name must refer to a client interface. If the given `clientComponent` refers to the component this interface belongs; this name must refer to a server interface.
`serverComponent`	the component that contains the server interface that has to be bound. If `null` the server component is assumed to be the component this interface belongs.
`serverItfName`	the name of the interface at the server side of the binding. If the given `serverComponent` is a sub component; this name must refer to a server interface. If the given `serverComponent` refers to the component this interface belongs; this name must refer to a client interface.
return	`0` if the binding has been added correctly. `FRACTAL_API_ILLEGAL_BINDING` if the binding cannot be created. `FRACTAL_API_NO_SUCH_SUB_COMPONENT` if the `clientComponent` or the `serverComponent` is neither a sub component nor the component this interface belongs. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if this component has a LifeCycleController interface, but it is not in an appropriate state to perform this operation. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int removeFcSubBinding(in fractal.api.Component clientComponent,
  in string clientItfName);
```

The `removeFcSubBinding` method removes a binding inside this component.

clientComponent	the component that contains the client interface that has to be unbound. If `null` the client component is assumed to be the component this interface belongs.
clientItfName	the name of the interface that has to be unbound. If the given `clientComponent` is a sub component; this name must refer to a client interface. If the given `clientComponent` refers to the component this interface belongs; this name must refer to a server interface.
return	`0` if the binding has been removed correctly. `FRACTAL_API_ILLEGAL_BINDING` if the binding cannot be removed. `FRACTAL_API_NO_SUCH_SUB_COMPONENT` if the `clientComponent` is neither a sub component nor the component this interface belongs. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if this component has a LifeCycleController interface, but it is not in an appropriate state to perform this operation. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

7.2.4. The `fractal.api.AttributeController` interface

This interface use constants defined in `fractal/api/ErrorConst.idt`.

This interface defines the following type:

```
enum AttributeType {
  INT_ATTR_TYPE,
  STRING_ATTR_TYPE,
  UINT8_ATTR_TYPE,
  INT16_ATTR_TYPE,
  UINT16_ATTR_TYPE,
  INT32_ATTR_TYPE,
  UINT32_ATTR_TYPE,
  INT64_ATTR_TYPE,
  UINT64_ATTR_TYPE,
  INTPTR_ATTR_TYPE,
  UINTPTR_ATTR_TYPE
};
```

The `enum AttributeType` enumerates the available attribute types.

This interface defines the following methods:

```
int listFcAttributes(in const string attributeNames[]);
```

The `listFcAttributes` method returns the names of the attributes of the component to which this interface belongs. More precisely, if the given `attributeNames` is `null`, this method returns the number of attributes. If it is not `null`, this method assumes that the given array is big enough to contain all the attribute names.

attributeNames	an array into which attribute names are copied. Can be `null`.
return	the number of attributes; or `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

```
int getFcAttribute(in const string attributeName, out void *value);
```

The `getFcAttribute` method returns the value of an attribute.	
`attributeName`	the name of the attribute to return.
`value`	(out parameter) the value of the attribute.
return	`0` if the operation succeed. `FRACTAL_API_NO_SUCH_ATTRIBUTE` if there is no such attribute.

`int getFcAttributeSize(in const string attributeName);`	
The `getFcAttributeSize` method returns the size of an attribute (in byte).	
`attributeName`	the name of the attribute to return.
return	the size if the attribute or `FRACTAL_API_NO_SUCH_ATTRIBUTE` if there is no such attribute.

`int getFcAttributeType(in const string attributeName, out enum AttributeType type);`	
The `getFcAttributeType` method returns the type of an attribute.	
`attributeName`	the name of the attribute to return.
`type`	(out parameter) the type of the attribute.
return	`0` if the operation succeed. `FRACTAL_API_NO_SUCH_ATTRIBUTE` if there is no such attribute.

`int setFcAttribute(in const string attributeName, in void *value);`	
The `setFcAttribute` method sets the value of an attribute.	
`attributeName`	the name of the attribute to set.
`value`	the value of the attribute to set.
return	`0` if the operation succeed. `FRACTAL_API_NO_SUCH_ATTRIBUTE` if there is no such attribute. `FRACTAL_API_ILLEGAL_ATTRIBUTE` if the attribute is not settable. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

7.2.5. The `fractal.api.LifeCycleController` interface

This interface use constants defined in `fractal/api/ErrorConst.idt`.

This interface defines the following methods:

`int getFcState();`	
The `getFcState` method returns the execution state of the component to which this interface belongs.	
return	`0` if the component to which this interface belongs is stopped; `1` if the component is started. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

`int startFc();`

The `startFc` method starts the component to which this interface belongs.	
return	`0` if the operation succeed. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if it fails.

`int` `stopFc();`	
The `stopFc` method stops the component to which this interface belongs. The result of a method call on a stopped component is undefined, except on its control interfaces (these calls are executed normally).	
return	`0` if the operation succeed. `FRACTAL_API_ILLEGAL_LIFE_CYCLE` if it fails. `FRACTAL_API_OPERATION_NOT_SUPPORTED` if this operation is not supported.

7.2.6. The `fractal.api.Factory` interface

This interface use constants defined in `fractal/api/ErrorConst.idt`.

This interface defines the following methods:

`int` `newFcInstance(out` **`void`**`* instance);`	
The `newFcInstance` method creates a new component.	
`instance`	(out parameter) the created instance.
return	`0` if the new instance has been created correctly. `INSTANTIATION_ERROR` if the instantiation fails.

`int` `destroyFcInstance(in` **`void`**`* instance);`	
The `destroyFcInstance` method destroys a previously created component.	
`instance`	the component instance to destroy.
return	`0` if the new instance has been created correctly. `ILLEGAL_LIFE_CYCLE` if the component has a `LifeCycleController` interface, but it is not in an appropriate state to perform this operation. `INSTANTIATION_ERROR` if the destroy fails.

7.2.7. `fractal/api/ErrorConst.idt`

This file defines the following constants:

```
#define FRACTAL_API_OK                         0
#define FRACTAL_API_INVALID_ARG               -1
#define FRACTAL_API_OPERATION_NOT_SUPPORTED -2
#define FRACTAL_API_NO_SUCH_INTERFACE         -3
#define FRACTAL_API_CLIENT                     0
#define FRACTAL_API_SERVER                     1

#define FRACTAL_API_ILLEGAL_BINDING          -10

#define FRACTAL_API_NO_SUCH_ATTRIBUTE        -30
#define FRACTAL_API_ILLEGAL_ATTRIBUTE        -31

#define FRACTAL_API_ILLEGAL_LIFE_CYCLE       -40
#define FRACTAL_API_STOPPED                    0
#define FRACTAL_API_STARTED                    1
```

7.3. The `memory.api` package

7.3.1. The `memory.api.Allocator` interface

A common interface to allocate and free memory space.

This interface defines the following methods:

`void *alloc(int size);`	
This method allocates `size` bytes and returns a pointer to the allocated memory. The memory is not cleared. The value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or `NULL` if the request fails.	
`size`	the amount of memory requested (in byte).
return	pointer to the allocated memory.

`void free(void *addr);`	
This method frees the memory space pointed to by `addr`, which must have been returned by a previous call to `alloc`. Otherwise, or if free has already been called before with the same `addr`, the behavior is undefined.	
`addr`	the pointer to be freed. May be `NULL`, in which case this operation does nothing.

7.4. The `memory` package

7.4.1. The `memory.AllocatorType` component type

A Type definition for memory allocator components. Provides a single interface called `allocation` whose type is `memory.api.Allocator`.

7.4.2. The `memory.Malloc` component

A simple memory allocator component that use `malloc` and `free` system calls to allocate/free memory.

7.5. The `boot` package

7.5.1. The `boot.Main` interface

The `boot.Main` IDL defines a `main` method. A component-based application must provides an interface of this type, so that it can be launched by a bootstrap compoenent.

This interface defines the following method:

`int main(int argc, char **argv);`	
Application entry-points	
`argc`	Argument count
`argv`	Argument values

return	Return value. By convention '0' means OK, any other value is considered as an error.

7.5.2. The `boot.ApplicationType` component type

An ADL type that can be extended by application components (i.e. components providing an `entryPoint` interface of type `boot.Main`).

7.5.3. The `boot.Bootstrap` component

The `boot.Bootstrap` primitive component is used to initialize and launch a component-based application on a POSIX system.

It requires an `entryPoint` interface that is intended to be bound to the `boot.Main` provided interface of the component-based application. It also requires a optional `appLCC` interface. This interface can be bound to the `fractal.api.LifeCycleController` interface of the application, if this latter provides it.

The component implementation contains the definition of the traditional "main" C function that is executed when the application is launched.

This component is singleton.

7.5.4. The `boot.GenericApplication` template component

This template ADL defines a generic top-level architecture made of a `boot.Bootstrap` component and a generic `App` component bound together.

The generic `App` component must conforms to the `boot.ApplicationType`.

This component is singleton.

7.5.5. The `boot.LCCApplicationType` component type

An ADL type that can be extended by application components that also have a `fractal.api.LifeCycleController` interface.

7.5.6. The `boot.GenericLCCApplication` template component

This template ADL defines a generic top-level architecture made of a `boot.Bootstrap` component and a generic `App` component bound together.

The generic `App` component must conforms to the `boot.LCCApplicationType`.

This component is singleton.

Appendix A. Glossary

Glossary

ADL (Architecture Description Language)
: The language used to describe the architecture of an application as a set of interconnected components.

CPL (Component Programming Language)
: The language on top of the C-Language that defines special constructs to capture component-based notions inside C programs.

fully-qualified-name
: The unambiguous name of an architecture definition or an interface signature. A fully-qualified-name is mane of two parts separated by a dot (`'.'`) : the *package-name* and the *simple-name*. The package-name may be empty (which correspond to the *default package*); in that case, the fully-qualified-name is simply the simple-name (without the dot).

IDL (Interface Definition Language)
: The language used to define the signature of the component interfaces.