USER MANUAL

Nomadik Multiprocessing Framework

Component Programming Model

NMF version 1.8 and more

April 2009

Contents

1 NMF Component model **7**
- 1.1 Separation of concerns . . . 7
  - 1.1.1 Separation of interface and implementation . . . 7
  - 1.1.2 Enforcing Data encapsulation . . . 8
  - 1.1.3 Component oriented programming . . . 8
  - 1.1.4 Inversion of control . . . 8
- 1.2 Components . . . 8
- 1.3 Interfaces . . . 9
- 1.4 External component structure . . . 10
  - 1.4.1 Local name . . . 10
  - 1.4.2 Role (server or client) . . . 11
  - 1.4.3 Contingency (mandatory or optional) . . . 11
  - 1.4.4 Cardinality (singleton or collection) . . . 11
- 1.5 Internal component structure . . . 12
- 1.6 Bindings . . . 12
  - 1.6.1 Primitive bindings . . . 13
  - 1.6.2 Composite bindings . . . 13
  - 1.6.3 Building bindings . . . 14
- 1.7 Attributes . . . 14
- 1.8 Properties . . . 15

2 Component model lifecycle **17**
- 2.1 Constructor and destructor . . . 17
- 2.2 Start and Stop . . . 17

3 Interface Description Language **19**
- 3.1 Interface declarations . . . 19
  - 3.1.1 Constant specifications . . . 20
  - 3.1.2 Naming conventions . . . 20
  - 3.1.3 Grammar specification of IDL . . . 20
  - 3.1.4 Grammar specification of a method declaration . . . 21

3.1.5 Type grammar specifications . . . 22
3.1.6 Examples . . . 24
3.2 Shared declaration types . . . 26
3.2.1 Grammar specification . . . 26
3.2.2 Structure declaration grammar specifications . . . 27
3.2.3 Enumeration grammar specifications . . . 28
3.2.4 Type definition grammar specifications . . . 29
3.2.5 Constant grammar specifications . . . 29
3.3 C/C++ mapping specifications . . . 30
3.3.1 Mapping for primitive types . . . 30
3.3.2 Mapping for arrays . . . 31
3.3.3 Mapping for pointers . . . 31
3.3.4 Mapping for structures . . . 31
3.3.5 Mapping for enumerations . . . 31
3.3.6 Mapping for type reference . . . 32
3.3.7 Mapping for methods . . . 32
3.3.8 Mapping for type names . . . 32

4 Architecture Description Language 33
4.1 Component specifications . . . 33
4.1.1 Naming conventions . . . 33
4.1.2 Constant specifications . . . 34
4.1.3 Component type grammar specifications . . . 34
4.1.4 Component grammar specifications . . . 35
4.2 Component types . . . 36
4.2.1 Grammar specifications . . . 36
4.2.2 Provided and required interfaces . . . 37
4.2.3 Properties . . . 39
4.2.4 Multiple inheritances . . . 39
4.2.5 Examples . . . 39
4.3 Primitive components . . . 40
4.3.1 Grammar specifications . . . 40
4.3.2 Attributes . . . 41
4.3.3 Primitive component sources . . . 42
4.3.4 Options . . . 43
4.3.5 Examples . . . 44
4.4 Composite components . . . 45
4.4.1 Grammar specifications . . . 45
4.4.2 Generic Composite . . . 46
4.4.3 Sub components . . . 47
4.4.4 Bindings . . . 48

4.4.5 Composite options . . . 49
4.4.6 Examples . . . 50

5 C/C++ Component Language 53
5.1 C/C++ enhancements . . . 53
5.1.1 Isolation and interfaces . . . 53
5.1.2 Source files . . . 53
5.1.3 Component framework wrapping . . . 53
5.1.4 Multiple instances . . . 54
5.1.5 Component compatibility . . . 55
5.2 Providing interfaces . . . 55
5.2.1 Declaring interface methods in C++ header file . . . 55
5.2.2 Duplicated methods . . . 56
5.2.3 Server interface collections . . . 57
5.2.4 Interrupt handlers . . . 58
5.2.5 Unambiguous interface . . . 59
5.3 Required interfaces . . . 59
5.3.1 Required interface collections . . . 60
5.3.2 Static required interface . . . 61
5.3.3 Optional required interface . . . 62
5.4 Bindings . . . 62
5.4.1 Static binding optimizations . . . 62
5.4.2 Interface delegation . . . 62
5.5 Legacy componentization . . . 63
5.5.1 Library calls . . . 63
5.5.2 Library implementations . . . 63

6 Build process 65
6.1 Compilation . . . 65
6.1.1 Compiler flags evaluation . . . 65
6.1.2 Overwriting binding component CFLAGS . . . 65
6.2 Errors . . . 66

Introduction

The Nomadik multiprocessing framework (NMF) programming model is a component-based approach for software design that shares concepts with many other programming models. The Nomadik toolset uses the NMF model to solve the problem of distributing software code execution between the ARM and the various digital signal processors (such as the MMDSP+) of the Nomadik platform. This problem is solved early in the software development process by providing tools for developing and debugging individual components, tuning their performances, and finally integrating the components in the finished application.

NMF provides the following advantages:

- provides a well-defined model for specifying sub-system software architecture
- allows the developer to extend, reconfigure and reuse code developed using the NMF principles
- runs the same code on each DSP (digital signal processor) by providing a unified run-time environment
- distributes intensive processing seamlessly over two or more DSPs
- hides distributed processing by transparently managing communications
- decreases memory footprint by providing dynamic loading and linking of components and enabling code to be shared between two instances of the same component implementation
- improves performance through dynamic configuration of components
- simplifies development by providing visual tools (using STWorkbench)
- improves security by managing authentication and access controls

Preface

Comments on this manual should be made by contacting your local ST Ericsson sales office or distributor.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- *definition*,
- `sample code`, `keyboard input` and `file name`,
- NAME of referenced project.

Software notation

Syntax definitions are presented in a modified BACKUS-NAUR FORM (BNF) unless otherwise specified.

- Terminal strings of the language, that is those not built up by rules of the language, are printed in double quote. For example, ``` ``void'' ```.
- Each phrase definition is built up using a double colon and an equals sign to separate the two sides (`::=`).
- Alternatives are separated by vertical bars (`|`).
- Optional sequences are enclosed in square brackets (`[` and `]`).
- An asterisk $*$ following any item indicates that it can appear zero or more times.
- Items that may be repeated appear in braces ({ and }).

Acknowledgements

Chapter 1

NMF Component model

The NOMADIK MULTIPROCESSING FRAMEWORK (NMF) is a modular and extensible component model for designing, implementing, deploying and reconfiguring various systems and applications.

The NMF component model is a refinement of a more general component model called FRACTAL [1] and a of a concrete implementation called THINK [2].

1.1 Separation of concerns

The NMF component model uses the "separation of concerns" design principle. The idea behind this principle is to separate the various concerns or aspects of a system into distinct pieces of code or runtime entities with as little overlap between these entities as possible. This makes the system configurable, secure and reliable.

The NMF component model exploits three specific features of the separation of concerns principle:

- separation of interface and implementation
- enforcing data encapsulation
- component oriented programming
- inversion of control

1.1.1 Separation of interface and implementation

This design pattern (also called "the bridge pattern") corresponds to the separation of the design and implementation. An application can be designed as a set of interfaces that fa-

[1] For more information, see http://fractal.objectweb.org
[2] For more information, see http://think.objectweb.org

cilitate communication between various components without any consideration having to be made as to how these components are to be implemented. Each component is designed as a self contained unit with a clearly defined function; therefore implementing individual components becomes a less complex task than implementing the same functionality using a conventional programming strategy.

1.1.2 Enforcing Data encapsulation

Encapsulation is an property of component design. It means that all of the component's data is contained and hidden in the component and access to it restricted to methods of that component. Encapsulation therefore guarantees in absence of fault memeory access the integrity of the data contained in the component.

The main benefit of encapsulation is that the programmer may change the implementation of the component without affecting the whole program, if he or she preserves the interface of the component. Any change of the data representation will affect only the implementation of the methods.

1.1.3 Component oriented programming

This pattern corresponds to the separation of the implementation concern into several composable, smaller concerns that are implemented in discrete, well-separated entities called *components*. A component is similar in concept to a class in object oriented programming, and many of the concepts of object oriented programming can also be applied to NMF. Within an application, one component can be replaced by another without any changes being necessary to other parts of the application. As long as it implements the required interfaces, an existing component can be re-used by a new application without having to be re-written.

1.1.4 Inversion of control

This pattern corresponds to the separation of function and configuration concerns. Instead of the components configuring themselves and being responsible for finding the resources that they need, an external configuration manager, running somewhere in the system and closely integrated with the operating system, configures and deploys NMF components directly and independently of each other.

1.2 Components

A component is a runtime entity and the basic unit of development and configuration. A component is composed of two parts:

- the *content* that manages the functionality of the component
- the *controller* that manages the non-functional concerns of the component (introspection[3], configuration, security, transactions, and so forth)

A component can contain a finite number of sub-components, nested to an arbitrary level. Sub-components may be shared with other components.

A *component template* is the definition of the data structure and the behavior (functionality) of a component, and can be used to create new components dynamically.

1.3 Interfaces

A component interacts with other components by invoking operations on its component interface. A component interface is an access point to a component. A component interface implements a language interface.

Note: In order to improve readability in this and other documents that discuss NMF, sentences such as "a component has a component interface that implements the language interface X" may be abbreviated to "a component has an interface X". It is important, however, not to confuse a component interface with a language interface.

- A *component interface* is an access point to a component. The component interface implements a language interface.
- A *language interface* is a type, that is, a set of structural properties that are common to more than one interface on multiple components. A language interface can be viewed as a template that is used to implement component interfaces.

An language interface is therefore a description of a set of methods without implementation. When a component implements a given interface type, each declared method belonging to that language interface is implemented for that particular component.

In order to access an interface, a binding must be established. A binding is a connection between the interfaces of two different components, see section 1.6.

In order to prevent hidden communication that can break down the component-based architecture, no interaction is permitted between components that does not use the implemented interfaces. For example, there are no shared variables between components. (It is possible, however, to gain read-only access to a component's attributes and properties; see section 1.7 and section 1.8 for more information.)

Interfaces are defined using the NMF interface description language (IDL), see chapter 3 for more information about the IDL. The structure of the components themselves (including the interfaces that they use) are defined using the NMF architecture description language (ADL). For more information about the ADL, see chapter 4.

[3] Introspection is the ability of a component to determine it's own status at run time.

1.4 External component structure

The only parts of a component that are visible to other components are its access points, called its external interfaces. Figure 1.1 gives a diagrammatic representation of a component with two server interfaces and one client interface. Everything else within the component is hidden from view.

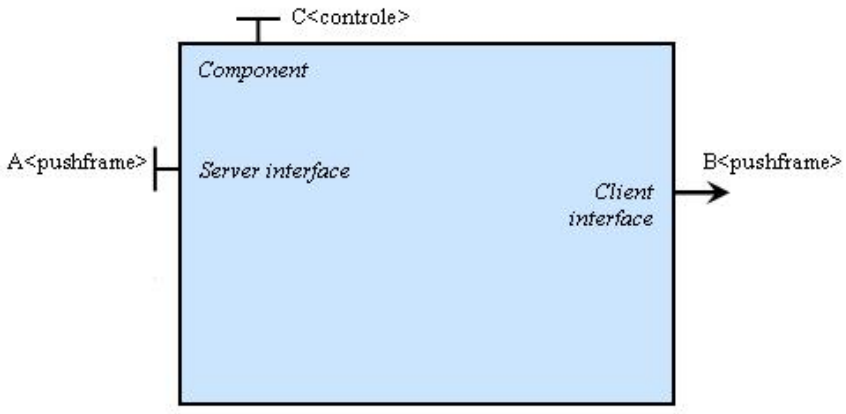

Figure 1.1: External view of a component

Each external interface is characterized by the following:

- local name
- role (server or client)
- contingency (mandatory or optional)
- cardinality (singleton or collection)

1.4.1 Local name

A local name distinguishes a given component interface from all other interfaces belonging to that component. A component can have more than one interface that implements the same language interface.

All the interfaces of a component must have distinct names. If the interface is part of a collection, then an index will be add to each entry in order to distinct them. However, two interfaces belonging to two different components may have the same name; this may be the case if two components fulfil similar functions and are interchangeable.

1.4.2 Role (server or client)

A *server interface* corresponds to access points that accept incoming method calls. To provide a service to another component, a component must possess a server interface. A server interface is also known as a provided interface, as the methods it defines are provided for use by other components.

A *client interface* corresponds to access points that support outgoing method calls. A component requiring a service from another component must possess a client interface. A client interface is also known as a required interface, as it represents an interface that requires methods that are defined in other components.

All client interfaces must be bound to their respective server interfaces before a component can be started, see chapter 2 for more information.

1.4.3 Contingency (mandatory or optional)

When a component runs, the operations of a *mandatory interface* are guaranteed to be available. This is always the case for a server interface, which must be available for any client that requests the service.

If an interface is an *optional interface*, then it is only active under certain circumstances, and there is no guarantee that the interface will be available. The developper must check the availability before calling the interface.

1.4.4 Cardinality (singleton or collection)

If a component has an interface that has a unique name, that interface has *singleton cardinality*.

If a component has more than one interface that share the same name, that interface has *collection cardinality* and we named it *collection interface*. An individual interface within a set of interfaces with collection cardinality is referenced using an index register in exactly that same way as an array is referenced in conventional programming languages, for example, `name[i]`, where `i` is the index.

The size of a collection can either be fixed statically when the component is first implemented, or it can be specified at load time. It cannot be changed when the component is actually running.

Collection interfaces are useful for components that have to provide a variable number of interfaces of the same type, such as an interrupt controller component.

1.5 Internal component structure

There are two types of component: *primitive components* and *composite components*.

A primitive component is the basic building block of an NMF system, and consists of a discrete and self contained unit written in the C or C++ programming language and compiled using the appropriate tools.

A composite component is constructed from a finite number of sub-components that are bound together in a manner that is appropriate to the functionality of that component. The sub-components can either be primitive components or other composite components. This recursive component model allows components to be nested at an arbitrary level. Each sub component has a local name inside the composite component in order to enable the composite component to identify it and manipulate it.

Figure 1.2 shows a simplified diagrammatic representation of the internal structure of a composite component.

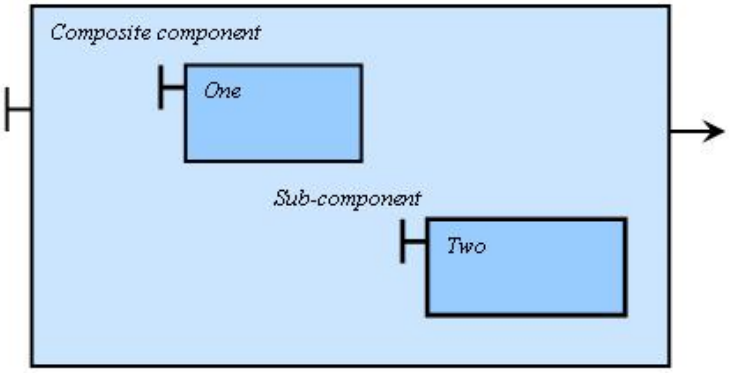

Figure 1.2: Internal view of a composite component

This composite component contains two sub-components, each of which has one server interface and one client interface.

1.6 Bindings

A *binding* is a communication path between component interfaces. A binding can be static, in which case it is defined prior to the compilation of the component and remains in place throughout the life time of the component, or it can be dynamic, in which case it is created when the application is deployed and may be different when the same component is used in different contexts.

Within a composite component, a sub-component can only be bound to another subcomponent or to the internal side of an external interface of the composite component, see Fig-

ure 1.3. A binding cannot cross the boundary of a composite component except through the means of an external interface of that component.

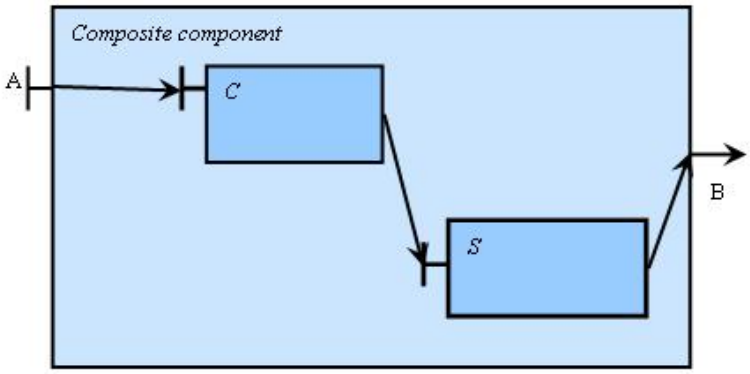

Figure 1.3: Binding of a composite component

There are two types of binding:

- primitive bindings
- composite bindings

1.6.1 Primitive bindings

A *primitive binding* is a permanent binding that exists between one client interface and one server interface of components that run on the same core. In many cases, a primitive binding may be implemented by nothing more sophisticated than a C function call in which the server interface defines a function and the client interface calls that function. A primitive binding is always synchronous, which means that the client interface's component halts until it receives a reply from the called method in the server interface.

A primitive binding can be established between a client and a server interface only if they have the same interface type.

It is permissible for several client interfaces to be bound to the same server interface with primitive bindings.

1.6.2 Composite bindings

A *composite binding* is a communication path that exists between an arbitrary number of component interfaces of arbitrary language types. These bindings are represented as a set of primitive bindings and *binding components* that potentially run on different cores within Nomadik (including the ARM). A binding component is a normal component (either primitive or composite) that is specifically dedicated to communication between other components. A

binding component may implement facilities such as FIFO queues and semaphores in order to manage asynchronous communication.

Figure 1.4 provides a diagrammatic representation of a composite binding.

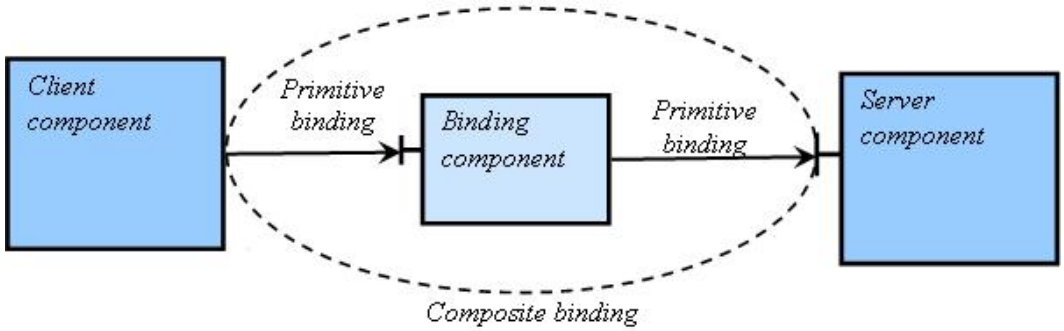

Figure 1.4: Composition of bindings

1.6.3 Building bindings

The NMF code generation tools can create binding components manually or automatically, as required by the design of the application. The many forms of composite binding that are available include the following:

- a binding between a component running on the host and a component running on a sub-system
- a binding between component running on a sub-system and a component running on the host
- a binding between component running on one sub-system and a component running on a different sub-system
- a binding for tracing every interaction between two components

Despite the apparent complexity, all of these connections are transparently created by NMF without the need to be manually developed. This includes marshalling and un-marshalling communication parameters.

1.7 Attributes

An *attribute* is an exported, dynamic variable configuration of a component. For example, a display size, an audio decoder stream frequency, and so on. Each attribute has a local name

that distinguishes it from other attributes. A component can have any number of attributes; these are generally of C/C++ primitive type.

NOTE. *NMF uses attributes to get the status of a component without having to define and use a binding. Only code running on the host processor has access to another component's attributes. This is achieved using the Component Manager API.*

1.8 Properties

A *property* is an exported, static constant configuration of a component. An example of a property is a decoder granularity. Each property has a local name that distinguishes it from other properties. A component can have any number of properties of string type.

NOTE. *NMF uses properties to define static information about a component without need to declare them on the sub-system side. Only code running on the host processor has access to a component's properties using the Component Manager API.*

Properties can be used manually or by the component manager. It is possible for a developer to specify a given property manually in order to manipulate the Component Manager (for example, to force code to run on a specific sub system).

Chapter 2

Component model lifecycle

This chapter describes the component model lifecycle, see Figure 5. Figure 5. Component model lifecycle To keep track of the component lifecycle, use the following interfaces in the component.

- lifecycle.constructor
- lifecycle.start
- lifecycle.stop
- lifecycle.destructor

The unique methods of these interfaces are called automatically if the lifecycle changes during either static or dynamic deployment.

2.1 Constructor and destructor

Constructor interface is called just after component memory loading.

Destructor interface is called just before component memory loading.

In this both interface implementation, it's not allowed to call required interface.

2.2 Start and Stop

Start interface is called when you start your component.

Stop interface is called when you stop your component.

In this both interface implementation, it's allowed to call required interface.

Chapter 3

Interface Description Language

NMF uses an *Interface Description Language* (IDL) as a standard protocol for describing the interfaces of all NMF components. The IDL used by NMF is a sub-set of the C programming language that only permits method, structure and enumeration declarations (and therefore resembles a C header file in structure). The grammar of IDL is specified in this chapter.

The NMF code generation tools use an interface description written using IDL to generate dedicated C code for implementing that interface. The generated code includes C declarations corresponding to the elements of the interface, and can include indirection components, such as stubs and skeletons, for communications.

Different components can share interface definitions, even if those components are not destined for the same application. For example, several different decoder implementations, (MPEG, H264 and so on) can share the same video decoder interface. Declarations can be shared between interfaces either by defining types within the interface description or in separate *Interface Description Type* (IDT) files (see section 3.2 for information about IDT files).

For legacy purpose, it's possible to declare *unmanaged interface* which allow legacy C header files to be included in interface. Thus, you don't have to declared your type in a IDT file. Nevertheless, unmanaged interface can't be used for asynchronous and distributed communications since type is not parser nor handle by tools.

Syntax definitions are presented in a modified Backus-Naur Form (BNF) unless otherwise specified, see Software notation on page 6.

3.1 Interface declarations

The definition of a given interface resides in an `.idl` file. To obtain the correct pathname for an interface file, convert all occurrences of "`.`" in the interface name to "`/`" and concatenate it with a `.idl` extension. For example, the interface specification of an interface called `api.bar` is found in a file with the file path `api/bar.idl`.

NOTE. *The old file extension* `.itf` *is deprecated. Use* `.idl` *instead.*

3.1.1 Constant specifications

In this grammar only positive integers are allowed.

```
INTEGER ::= DECIMAL | HEXADECIMAL | OCTAL
STRING ::= C_STRING
CHARACTER ::= C_CHARACTER
ConstantIdentifier ::= Identifier
```

`ConstantIdentifier` specifies the constant defined in the IDL/IDT file and used in miscellaneous type definitions, for example, arrays.

3.1.2 Naming conventions

An identifier uses a similar syntax to an identifier in C. In modified Backus-Naur notation, this is:

```
Identifier ::= Letter ( Letter | Digit )*
FullyQualifiedName ::= Identifier ( "." Identifier )*
```

Therefore `Interface`, `A23`, and `output1` are all permitted identifiers; `99A` is not.

3.1.3 Grammar specification of IDL

The IDL grammar is a sub-set of the grammar of the C programing language.

Listing 3.1 shows the grammar definition for an IDL declaration of an interface.

Listing 3.1: IDL grammar specification for an .idl file

```
InterfaceDeclaration ::=
    ( ``#'' ``include'' ``<'' IncludePathName ``>'' )*
    (
        StructDeclaration ``;'' |
        EnumDeclaration ``;'' |
        TypedefDeclaration ``;'' |
        DefineDeclaration
    )*
    ``interface'' InterfaceTypeName [ ``unmanaged'' ]
    ``{''
        ( MethodDeclaration ``;'' )*
    ``}''
    EOF

InterfaceTypeName ::= FullyQualifiedName

IncludePathName ::=
```

```
        Name ( ``/'' Name)* ``.idt'' |
        Name ( ``/'' Name)* ``.h''
```

Where:

include A sequence of include specifies the file where the external type used by this interface is defined. A file must be an `.idf` file that contains the interface description type or a `.h` file that contains legacy definition:

- Included IDT files are analyzed by the NMF parser, see section 3.2.
- Included H files are not analyzed and can be used only with unmanaged interface.

StructDeclaration A structure that is defined in the interface.

EnumDeclaration An enumeration that is defined in the interface.

TypedefDeclaration A type that is defined in the interface.

DefineDeclaration A sequence of DefineDeclaration specifies a list of constants defined in this interface.

unmanaged Specify that interface is unmanaged.

MethodDeclaration A method supplied by the interface, see subsection 3.1.4 for more information.

Any number of structures, enumerations and typedefs can be included in the `.idl` file. They are delimited with "`;`" and may be specified in any order. Any number of methods may be declared (also delimited with "`;`") but all method declarations must follow `struct`, `enum` and `typedef` declarations.

3.1.4 Grammar specification of a method declaration

Listing 3.2 shows the grammar of a method declaration. The grammar is similar to that used by the C programming language.

Listing 3.2: Method declaration grammar specification

```
MethodDeclaration ::=
    ResultType Identifier ``('' 
    [
        ``void'' |
        FormalParameter (``,'' FormalParameter )*
    ]
    ``)''
```

Where:

ResultType The returned type of a method. Some binding factory that can not handle return type can restrict such type to void type, for example asynchronous (distributed) communication, see subsection 3.1.5.

Identifier The name of the method.

FormalParameter A parameter that is to be passed to the method. A method can have zero or more parameters.

Listing 3.3 shows the grammar of a parameter declaration.

Listing 3.3: Parameter declaration grammar specification

```
FormalParameter ::=
    [ ``const'' ] ParameterType [ ``&'' ]
    Identifier
    ( ``['' ( INTEGER | ConstantIdentifier ) ``]'' )*
```

Where:

const A keyword that specifies that a parameter is a constant and that the called method cannot modify it.

ParameterType The type of the parameter, see subsection 3.1.5.

& This C enhancement is directly based on the C++ specification and enables minimizing copying in distributed communication. (The structure is only copied between cores and not between a client and a client stub or between a server stub and a server).

Identifier The name of the parameter.

INTEGER If a parameter is an array, the array length `INTEGER` or `ConstantIdentifier` enables the stub to copy the right number of elements. If the size is specified through a constant, this constant must be defined with a `#define`.

3.1.5 Type grammar specifications

The grammar of the types used in method declaration is shown in Listing 3.4:

Listing 3.4: Type grammar specification

```
ResultType ::=
    (
```

```
        PrimitiveType |
        ComplexType |
        ''void''
    )
    ( [ MemoryQualifier ] ''*'' )*

ParameterType ::=
    (
        PrimitiveType |
        ComplexType
    )
    ( [ MemoryQualifier ] ''*'' )*

PrimitiveType :: =
    ''t_uword'' | ''t_sword'' |
    ''t_physical_address'' |
    ''char'' |
    ''t_bool'' |
    ''t_uint64'' | ''t_sint64'' |
    ''t_uint56'' | ''t_sint56'' |
    ''t_uint48'' | ''t_sint48'' |
    ''t_uint40'' | ''t_sint40'' |
    ''t_uint32'' | ''t_sint32'' |
    ''t_uint24'' | ''t_sint24'' |
    ''t_uint16'' | ''t_sint16'' |
    ''t_uint8'' | ''t_sint8''

ComplexType ::=
    ''struct'' Identifier |
    ''enum'' Identifier |
    Identifier |
    ''void'' [ MemoryQualifier ] ''*''

MemoryQualifier ::= Identifier
```

Where:

PrimitiveType The type is a primitive type. Potentially, a primitive type that does not exist on the target processor will be refused by tools.

void As a return type, indicates that the method does not return the type. As a parameter type, indicates that the method does not require any parameters.

Identifier The name of the parameter.

struct The type is a structure. The structure must be declared in:

- the content of the interface
- in an included interface description type file

enum The type is an enumeration. Enumerations can be declared in:

- the content of the interface
- in an included interface description type file

typedef Specifies that the type is a type definition. A type definition can be declared in:

- the content of the interface
- in an included interface description type file

"*" Indicates that the identifier that follows this symbol is a pointer to a variable of the given type. "`void *`" indicates that the parameter identifier is untyped.

MemoryQualifier Specifies the memory qualifier of a memory. The qualifier depends on the C compiler and is not targeted to be used in distribution. The qualifier is interpreted by the programming model, see the C compiler manual for information.

3.1.6 Examples

The following example shows the client interface that provides a method for adding two integers, a and b:

```
interface adder {
    void add(t_uint32 a, t_uint32 b);
}
```

The interface used internally by the execution engine to send activity traces though the XTI is defined as:

```
interface XTI.tracer {
    void traceSched(t_uint8 id, t_uint24 this, t_uint24 address);
}
```

The example in Listing 3.5 defines an interface for managing an interrupt. Note that traditional C style comments are permitted within IDL.

Listing 3.5: IDL example 1

```
/*!
 * \brief IRQ controller API
 */
interface IRQ.Controller
{
    /*!
     * \brief Return irq number associated with this name
     * \param[in] name Symbolic name of an interruption.
     * \return Irq line corresponding to the interruption name.
     */
```

```
    t_uint24 getIrq(char* name);
    /*!
     * \brief Disable the irq.
     * \param[in] irq Irq line to be disabled.
     */
    void disable(t_uint24 irq);
    /*!
     * \brief Enable the irq.
     * \param[in] irq Irq line to be enabled.
     */
    void enable(t_uint24 irq);
}
```

NOTE. *Like shows in this example, you can used doxygen tags to comment your interface.*

Listing 3.6 is a more complex interface used in the audio framework to identify the format of a decoded stream.

Listing 3.6: IDL example 2

```
typedef enum {
    FS_ILLEGAL_KHZ,      //!< Illeqal frequency
    FS_192KHZ,           //!< 192000 Hz
    FS_176_4KHZ,         //!< 176400 Hz
    FS_128KHZ,           //!< 128000 Hz
    FS_96KHZ,            //!< 96000 Hz
    FS_88_2KHZ,          //!< 88200 Hz
    FS_64KHZ,            //!< 64000 Hz1
    FS_48KHZ,            //!< 48000 Hz
    FS_44_1KHZ,          //!< 44100 Hz
    FS_32KHZ,            //!< 32000 Hz
    FS_24KHZ,            //!< 24000 Hz
    FS_22_05KHZ,         //!< 22050 Hz
    FS_16KHZ,            //!< 16000 Hz
    FS_12KHZ,            //!< 12000 Hz
    FS_11_025KHZ,        //!< 11025 Hz
    FS_8KHZ,             //!< 8000 Hz
    FS_7_2KHZ,           //!< 7200 Hz
    NUM_SUPPORTED_FS     //!< Max supported frequency
} t_audio_frequency;

interface audio.decoder.notify
{
    void inform(
        t_audio_frequency sampleFrequency,
        t_uint16 sampleSize,
        t_uint16 nbChan);
}
```

Listing 3.7 is an example of unmanaged interface.

Listing 3.7: IDL example 3

```
interface ComplexComputer unmanaged
{
    ComplexType add(ComplexType a, ComplexType b);
    ComplexType inc(ComplexType a);
    ComplexType sub(ComplexType a, ComplexType b);
}
```

3.2 Shared declaration types

If an interface declaration definition is intended to be used by more than one interface, it can be placed in a separate file and imported into any given interface definition file by using a `#include` directive. The file containing importable interface definitions should always have a `.idt` (interface description type) extension. Any `.idl` file can import any number of `.idt` files.

3.2.1 Grammar specification

The grammar for an IDL declaration of an interface (within an `.idt` file) is slightly different to the grammar used for an `.idl` file, see Listing 3.8.

Listing 3.8: IDL grammar specification for an .idt file

```
InterfaceDeclaration ::=
    ``#ifndef'' Identifier
    ``#define'' Identifier
    ( ``#'' ``include'' ``<'' IdtPathName ``>'' )*
    (
        StructDeclaration ``;'' |
        EnumDeclaration ``;'' |
        TypedefDeclaration ``;'' |
        DefineDeclaration
    )*
    ``#endif''
    EOF
```

Where:

#ifndef, #define and #endif — Directives to enable conditional compilation of sections of IDL definitions. These directives operate in exactly the same manner as the C preprocessor directives with the same names.

FILENAME — Specify a file that defines external types used by the interface. Files are either.idt files that contain interface description types or `.h` C header definitions. Any included IDT files are analyzed by the NMF parser, see

There can be any number of `#include` directives in a file.

StructDeclaration A structure that is defined in the interface.

EnumDeclaration An enumeration that is defined in the interface.

TypedefDeclaration A type that is defined in the interface.

DefineDeclaration A sequence of DefineDeclaration specifies the list of constants defined in this interface.

Any number of structures, enumerations and typedefs can be included in the .idt file. They are delimited with ";" and may be specified in any order.

3.2.2 Structure declaration grammar specifications

The grammar of a structure declaration is:

```
StructDeclaration ::=
    ``struct'' Identifier ``{'' ( FormalField ``;'' )+ ``}''
```

Where:

Identifier The name of the structure.

FormalField A field (which is generally one of a sequence of fields).

The grammar of a field declaration is:

```
FormalField ::=
    FieldType Identifier
    (
        ``['' ( INTEGER | ConstantIdentifier ) ``]''
    )*
```

Where:

Identifier The name of the field.

FieldType The type of the field.

INTEGER If a field is an array or ConstantIdentifier, the array length INTEGER specifies the number of elements.

Listing 3.9 shows the grammar of a field type.

Listing 3.9: Field type grammar

```
FieldType ::=
    (
        PrimitiveType |
```

```
        ComplexType |
        ``struct'' ``{'' ( FormalField ``;'' )+ ``}'' |
        ``enum'' ``{'' Enumerator ( ``,'' Enumerator )* ``}'' |
    ) ( [ MemoryQualifier ] ``*'' )*
```

Where:

PrimitiveType	The field is a primitive type.
ComplexType	The field is a complex type. Structure, enumeration and type definitions declared outside the interface in a C header file are not permitted, see Section 3.1.5: Type grammar specifications on page 18.
struct	The type of the field is an embedded structure.
enum	The type of the field is an embedded enumeration.
"*"	Indicates that the object being referenced is untyped.

3.2.3 Enumeration grammar specifications

The grammar of an enumeration declaration is:

```
EnumDeclaration ::=
    ``enum'' Identifier ``{'' Enumerator ( ``,'' Enumerator )* ``}''
```

Where:

Identifier	The name of the enumeration.
Enumerator	A constant in the enumeration.

The grammar of an Enumerator constant declaration is:

```
Enumerator ::=
    Identifier
        [ ``='' (
            [``-'' | ``+''] INTEGER |
            CHARACTER |
            ConstantIdentifier ) ]
```

Where:

Identifier	The name of the constant.
INTEGER and ConstantIdentifier	Specifies the value of the enumerator, which could be another enumerator.

3.2.4 Type definition grammar specifications

The grammar of a type definition declaration is:

```
TypedefDeclaration ::=
    ''typedef'' FieldType Identifier
```

Where:

Identifier The name of the type definition.

FieldType The type of the definition.

3.2.5 Constant grammar specifications

Grammar of a constant definition declaration:

```
DefineDeclaration ::=
    ''#define'' ConstantIdentifier (
        [ ''-'' | ''+'' ] <INTEGER> |
        <STRING> |
        <CHARACTER>
)
```

Where:

ConstantIdentifier Specifies the name of the constant, for example, to define the size of an array.

INTEGER, STRING or CHARACTER Specifies the value of the constant.

Constants are directly redefined like declaration in generated files.

Examples

Listing 3.10 shows an example of an .idt file that defines the interface that is used for unary testing of parameter marshalling. It imports an `.idt` file called `binding.idt`.

Listing 3.10: IDT example

```
#include <binding.idt>
void callInt8(t_uint8 int8);
void callInt16(t_uint16 int16);
void callInt24(t_uint24 int24);
void callInt32(t_uint32 int32);
void callWord(t_uword word);
void callPointer(t_uint32 *int32 );
void callArrayInt8(t_uint8 tab[4]);
```

```
void callArrayInt16(t_uint16 tab[4]);
void callArrayInt24(t_uint24 tab[4]);
void callArrayInt32(t_uint32 tab[4]);
void callStruct(t_struct foo);
void callEnum(t_color color);
void callRef(t_struct &foo);
void callMultiArgs(t_uint8 int8,
        t_uint16 int16,
        t_uint8 int8_2,
        t_uint32 int32);
```

The `#include` file, `binding.idt`, is shown in Listing 3.11.

Listing 3.11: IDT example included file

```
#ifndef __BINDING_IDT
#define __BINDING_IDT
typedef enum
    {
        BLUE, YELLOW, RED
    } t_color;

    typedef struct
    {
        t_uint8 fieldOctet1;
        t_uint16 field16;
        t_uint32 field32;
    } t_struct;
#endif
```

Note the use of a `#ifndef` statement to ensure that the include file is not included more than once.

3.3 C/C++ mapping specifications

When the NMF tools generate C/C++ code based upon an IDL definition, they map the IDL types to the equivalent C/C++ language types. Each of the different varieties of core found in a Nomadik chip use a slightly different definition of types; the NMF tools ensure that the types used are correct for the particular core on which the component is expected to run.

Where a given component is designed to run on more than one type of core, then that component is cross-compiled to create individual binary versions that are specific to a given core. The source code of the component itself remains unchanged.

3.3.1 Mapping for primitive types

For each core, the IDL mapping specifies how primitive types are transformed by tools and handled in C/C++ language. This mapping is shown in Table 3.1.

Table 3.1: Mapping for primitive types

Primitive type	32bits (ARM, ...)	16bits (MMDSP, ...)	24bits (MMDSP, ...)
t_[u,s]word	[un]signed int	[un]signed int	[un]signed int
t_[u,s]int64	[un]signed long long	N.A.	N.A.
t_[u,s]int56	[un]signed long long	N.A.	[un]signed long long
t_[u,s]int48	[un]signed long long	N.A.	[un]signed long
t_[u,s]int40	[un]signed long long	[un]signed long long	[un]signed long
t_[u,s]int32	[un]signed long	[un]signed long	[un]signed long
t_[u,s]int24	[un]signed long	[un]signed long	[un]signed int
t_[u,s]int16	[un]signed short	[un]signed int	[un]signed int
t_[u,s]int8	[un]signed char	[un]signed char	[un]signed char
char	char	char	char

3.3.2 Mapping for arrays

IDL arrays map directly to C/C++ arrays. All array indices run from `0` to `<size - 1>`.

Type	C/C++ member mapping	C/C++ parameter mapping
t_uint24 a[LENGTH]	t_uint24 a[LENGTH]	t_uint24 *a

3.3.3 Mapping for pointers

IDL pointers directly map to C/C++ pointers.

Type	C/C++ member mapping	C/C++ parameter mapping
t_uint24* a	t_uint24* a	t_uint24* a

3.3.4 Mapping for structures

IDL structures map directly onto C structures. The members all remain unchanged, but member types are converted according to the mapping for a primitive type.

Type	C/C++ member mapping	C/C++ parameter mapping

struct structure {
t_uint24 x;
}
	struct structure a	struct structure a

3.3.5 Mapping for enumerations

IDL enumerations map directly to C/C++ enumerations.

Type	C/C++ member mapping	C/C++ parameter mapping
enum enumeration { A, B, C }	enum enumeration a	enum enumeration a

3.3.6 Mapping for type reference

In C

Mapping as for a pointer. C does not handle a reference type as a parameter.

Type	C member mapping	C parameter mapping
struct structure &	N.A	struct structure*

In C++

Type	C++ member mapping	C++ parameter mapping
struct structure &	N.A	struct structure &

3.3.7 Mapping for methods

When multiple instances are managed by the compilation tool chain (which is the case when considering sub-systems), IDL methods map directly to C methods.

A `this` parameter is available to pass the component context to the method; `this` is always the first parameter in a method declaration.

In the component source code, the implementation of a method is encapsulated in the `METH` macro. The C preprocessor expands the `METH` macro to generate the appropriate C code for the method.

Method	Host mapping	Sub-system mapping
void bar()	void METH(bar)(void* this)	void METH(bar)() or void bar()

3.3.8 Mapping for type names

To be done

Chapter 4

Architecture Description Language

NMF uses an architecture description language (ADL) to define the structure of a component. This is done by specifying the interfaces that belong to that component and indicating which interfaces it provides as client interfaces and which it requires as server interfaces.

Component types (which act as templates for designing new components that share specific features or functionality) are also defined using ADL.

When defining a primitive component, the developer uses the ADL definition to indicate the location of the C/C++ source code for that component. For a composite component, the developer uses ADL to identify the sub components (each of which have their own ADL definition) and to define the bindings between the interfaces of those sub components.

ADL can also pass some compilation options relating to the component to the compilation tool chain, as well as define specific properties that can be read by the component manager.

ADL has a different grammar to IDL; the grammar of ADL is specified in this chapter.

4.1 Component specifications

This section describes the NMF naming conventions and component grammar specifications.

4.1.1 Naming conventions

The names of components, component types and interface types are designed using the following naming scheme.

```
Name ::= Letter ( Letter | Digit )*
FullyQualifiedName ::= Name ( ``.'' Name )*
```

Where:

FullyQualifiedName This is the name of the component, component type or inter-

face type. This name and the name of the file storing the description are correlated; see subsection 4.1.3 for more details.

4.1.2 Constant specifications

In this grammar only positive integer are allowed.

```
INTEGER ::= DECIMAL | HEXADECIMAL | OCTAL
STRING ::= C_STRING
CHARACTER ::= C_CHARACTER
```

4.1.3 Component type grammar specifications

A component type declaration must reside in a file with a .type extension with a file pathname obtained by replacing occurrences of "." within the component type name with "/". For example, the definition of a component type named `example.bar` is found in the file named `example/bar.type`.

The grammar for an ADL declaration of a component type is shown in Listing 4.1.

Listing 4.1: Grammar specification for a component type file

```
TypeDeclaration ::=
        ''type'' TypeNameA
            [ ''extends'' TypeNameB ( '','' TypeNameB )*]
        ''{''
            ComponentTypeDeclaration
        ''}''
        EOF

TypeNameA ::= FullyQualifiedName

TypeNameB ::= FullyQualifiedName
```

Where:

type Keyword that specifies that this is a component type definition.

TypeNameA The name of the component type being defined.

TypeNameB If this type definition extends one or more existing component types, then this is the name of a component type that TypeNameA is extending.

ComponentTypeDeclaration The declaration of the component type structure, see subsection 4.1.4 for more details.

4.1.4 Component grammar specifications

All files relating to the definition of a component (including source code) resides in a specified directory. The file path for this directory is determined by converting occurrences of "`.`" in the component name to "`/`". For example, the files that define a component named `example.foo` is found in the directory named `example/foo`. If the component is a primitive component, this directory contains sub-directory called `src` that contains the C/C++ code for the component.

In a C project, the extension of source code files must be `.c` for C and `.s` for assembler. In a C++ project, the extension of source code files must be `.cpp` for C++ and `.s` for assembler. You can mix C and C++ component in the same project compiled with C++.

A component declaration itself resides in a file with an `.adl` extension. To obtain the filename for this path, take the final element of a component template name and concatenate it with "`.adl`". This file must reside at the root of the component directory. For example, the component specification of the component named `example.foo` is found in the file named `example/foo/foo.adl`.

In order to avoid renaming, `example/foo/component.adl` is also recognized and accepted by toolbox, where the file `component.adl` must reside at the root of the component directory.

NOTE. *The old file extension `.conf` is deprecated. Use `.adl` instead.*

The grammar for an ADL declaration of a component is shown in Listing 4.2.

Listing 4.2: Grammar specification for a component

```
ComponentDeclaration ::=
   (
      ''primitive'' ComponentName
         [ ''extends'' TypeName ( '','' TypeName )* ]
         [ ''singleton'' ]
      ''{''
         PrimitiveDeclaration
      ''}''
   ) | (
      ''composite'' CompositeNameDeclaration
         [ ''extends'' TypeOrCompositeDefinition
               ( '','' TypeOrCompositeDefinition )* ]
         [ ''singleton'' ]
      ''{''
         CompositeDeclaration
      ''}''
   )
   EOF

ComponentName ::= FullyQualifierName

TypeOrCompositeDefinition ::= TypeName | CompositeDefinition
```

Where:

primitive or composite Keywords that specify if the component is a primitive component or a composite component. If neither keyword is provided, it is assumed that the component being defined is a primitive component.

ComponentName The name of the component being defined. If neither the primitive or composite keywords have been specified, this is not required.

TypeName If this component definition extends the definition of one or more existing component types, then this specifies the component type that is being extended.

singleton A singleton can instantiate one time only on one same MPC and share between several clients running on the same MPC.

Singleton are useful for static components such as drivers or for shared libraries. The infrastructure ensures that if a client require two instantiations, only one is instantiated to return the same reference. Due to the reference counter, both must be the same number in order to destroy the singleton.

CompositeNameDeclaration Specifies a generic composite declaration. See subsection 4.4.2 for more information.

CompositeDefinition Specifies a specialized generic composite declaration. See subsection 4.4.2 for more information.

4.2 Component types

Component definitions are defined by extending an existing component type. The mechanism is analogous to class inheritance in object oriented programming; that is, a new component definition adds new elements or overrides existing elements (such as server interfaces, client interfaces and properties) inherited from the component type it is extending. A developer can use this mechanism to define new components as sub definitions of existing abstract (and possibly non-implemented) component definitions.

In the current version of NMF, only overriding properties is permitted, where the existing value is replaced by a new value. Function overloading is not permitted.

4.2.1 Grammar specifications

The grammar of a component type declaration is shown in Listing 4.3.

Listing 4.3: Component type grammar specification

```
ComponentTypeDeclaration ::=
        (
            InterfaceDeclaration |
            PropertyDeclaration
        )*
```

Where:

InterfaceDeclaration Specifies an interface for the component type. The declarations can be specified in any order, and server and client interface declarations may be interleaved.

PropertyDeclaration Specifies a property for the component type. A property is a constant that is read from the host through the component manager only.

4.2.2 Provided and required interfaces

The grammar for an interface declaration is shown in Listing 4.4.

Listing 4.4: Interface grammar specification

```
InterfaceDeclaration ::=
   (
      ''provides'' InterfaceType ''as'' InterfaceName
         [ ''['' CollectionSize '']'' ]
         [ ''unambiguous'' ]
         [ ''prefix'' MethodPrefix ]
         [ ''interrupt'' InterruptLine ]
         [ ''referenced'' ]
   ) |
   (
      ''requires'' InterfaceType ''as'' InterfaceName
         [ ''['' CollectionSize '']'' ]
         [ Contingency ]
   )

InterfaceType::= FullyQualifierName
InterfaceName ::= Name
CollectionSize ::= INTEGER
MethodPrefix ::= Identifier
InterruptLine ::= INTEGER
Contingency ::= ''optional'' | ''mandatory'' | ''static''
```

Where:

provides or requires The keyword `provides` specifies that the interface is a provided (that is, a server) interface. The keyword `requires`

specifies that the interface is a required (that is, a client) interface.

InterfaceType The fully qualified name of the interface type. This is an interface that is specified in IDL in its own file.

InterfaceName The local name of this interface in the context of the component being defined.

CollectionSize Specifies that this interface is a collection and also indicates the size of the collection. The collection size must be in range $[1...255]$.

Use the format (either in ADL or in C/C++): `InterfaceName[index]` to reference a specific interface from the collection, where $index$ is in the range 0 to $CollectionSize - 1$. Only a static declaration of collection size is permitted.

unambiuous Specifies that developer assume that no name conflict could occur between interface name symbol and other symbols.

referenced Specifies that the provided interface descriptor was used internally by source code (for example to pass it to another component through interface delegation) and thus tools need to generate it.

Contingency Specifies whether a required interface is `optional`, `mandatory` or `static` (implies `mandatory`). An optional interface can remain unbound (that is, not used by the component implementation).

A static required interface is an interface that is called statically with `meth` instead of `I.meth`. If not specified, an imported interface is considered as mandatory. A static required interface binds only to a singleton component and for optimization purpose because switch context is not required when calling singleton component or for legacy purpose to call shared library.

prefix If the component contains interfaces that declare methods with the same name, use `prefix` to distinguish the methods belonging to this interface from any identically named methods in any other interfaces. For example, if you specify a prefix of A for an interface that supplies a method called `foo()`, the NMF code generation tools rename the method as `Afoo()`. Because the purpose of `prefix` is to make the code easy to understand, `prefix` can not be specified in a composite component that does not contains sources.

interrupt Specifies that this interface must be called when the interrupt `InterruptLine` occurs. The interface type of this interface must be `exception.handler`, which is a predefined interface type. `exception.handler` contains only one method named `handler`; this method should be implemented as a handler for the interrupt.

4.2.3 Properties

Listing 4.5 shows the grammar for a property declaration.

Listing 4.5: Property grammar specification

```
PropertyDeclaration ::=
        ``property'' PropertyName ``='' PropertyValue

PropertyName ::= Name
PropertyValue ::=
        [``-'' | ``+''] INTEGER |
        STRING
```

Where:

PropertyName The name of the property.

PropertyValue Specifies the value of the option that can be an integer (decimal, hex or octal) or a string.

4.2.4 Multiple inheritances

Multiple inheritance is permitted. A component that inherits from more than one parent component type inherits all the entities from each parent.

To avoid confusion, extending two component types that contain entities with the same name is not permitted. This applies to required interfaces, provided interfaces or properties.

4.2.5 Examples

Listing 4.6 shows the definition of a component type called `audio.codec` (in the file `audio/codec.type`). This component type is an audio decoder in the audio framework. The definition lists the interfaces that a component based on this component type requires and provides, and defines properties called `inputGranularity` and `outputGranularity`.

Listing 4.6: ADL example 1

```
type audio.codec
{
```

```
    // Inform
    requires audio.codec.inform as outputInform

    // Input bit-stream
    provides audio.link.bs.configbs as inputConfig
    requires audio.link.bs.streamctrl as inputControl
    provides audio.link.bs.streaminput as inputPort
    property inputGranularity = 1

    // Output PCM stream
    requires audio.link.pcm.streaminput as outputPort
    provides audio.link.pcm.streamctrl as outputControl
    property outputGranularity = 1
}
```

The "as" clause ensures that each interface is given a unique name wherever it is implemented, even though different interfaces can use the same template.

The edge bit-stream binding component supplies the audio.link.bs.* interface.

The edge pcm binding component supplies the audio.link.pcm.* component.

4.3 Primitive components

A primitive component is defined by identifying its interfaces, its attributes and properties, and the location of the C/C++ source code that provides the component's functionality.

4.3.1 Grammar specifications

Listing 4.7 shows the grammar of a primitive component declaration.

Listing 4.7: Primitive component grammar specification

```
PrimitiveDeclaration ::=
    (
        InterfaceDeclaration |
        AttributeDeclaration |
        PropertyDeclaration
    )*
    ( SourceDeclaration )*
    ( OptionDeclaration )*
```

Where:

InterfaceDeclaration Specifies the interfaces of the component. The order in which interfaces are declared is irrelevant; provided (server) and required (client) interface declarations may be interleaved.

AttributeDeclaration An attribute exported by the component. An attribute is a C/C++ variable that can be read from the host through the component manager API.

PropertyDeclaration A property exported by the component. A property is a constant that can be read from the host through the component manager.

SourceDeclaration The sources that are needed to build the component. If no source is specified, the NMF tools use the entire contents of the src sub directory. (See subsection 4.3.3 for more details.)

OptionDeclaration Component compilation options. This is a compilation option that can modify the compilation behavior for the component.

4.3.2 Attributes

The grammar of an attribute declaration is shown in Listing 4.8.

Listing 4.8: Attribute component grammar specification

```
begin{lstlisting}
AttributeDeclaration ::=
    ``attributes''
        (
            PrimitiveType AttributeName
                  [ ``='' ( AttributeValue | ParameterName ) ]
            |
            AttributeName
        )
AttributeName ::= Name

PrimitiveType :: =
    (
        ``t_bool'' |
        ``t_uword'' | ``t_sword'' |
        ``char'' |
        ``t_uint64'' | ``t_sint64'' |
        ``t_uint56'' | ``t_sint56'' |
        ``t_uint48'' | ``t_sint48'' |
        ``t_uint40'' | ``t_sint40'' |
        ``t_uint32'' | ``t_sint32'' |
        ``t_uint24'' | ``t_sint24'' |
        ``t_uint16'' | ``t_sint16'' |
        ``t_uint8'' | ``t_sint8''
    ) [ ``*'' ]

AttributeValue ::=
    [``-'' | ``+'' ] INTEGER |
    CHARACTER |
    STRING
```

Where:

AttributeName Specifies the local name of an attribute in the component. In C/C++ code, use this attribute encapsulated in the `ATTR` macro. For example, if the component declares a `bar` attributes, to access the attribute, use `ATTR(bar)`. In C++, you can also directly use the attribute name without need the `ATTR` macro.

PrimitiveType Specifies the type of the attribute. NMF automatically declares the attributes. If not specified, the type is anonymous and is not manageable by NMF and must be declared in the C code. Depending on the MPC architecture, some primitive type can not be dynamically read through the component manager. Note the initial value can not be set to have an anonymous attribute type either directly during declaration or through assign.

AttributeValue The initial value of an attribute, setting an initial value is permitted only if the type is specified.

ParameterName Specifies that the initial value must be passed through composite attribute parameters. See subsection 4.4.2 for more information.

4.3.3 Primitive component sources

The grammar for a source declaration is shown in Listing 4.9.

Listing 4.9: Primitive component source grammar specification

```
SourceDeclaration ::=
    ``sources'' PathName
        [ CFlags ]
PathName ::= Name ( ``/'' Name)* ``.'' Name
CFlags ::= 'STRING
```

Where:

PathName The path to the source relative to the component directory. By default, the NMF compilation tools search for component sources in the `src` directory of the component source directory. For example, the source of the component `example.bar` is found in the directory

CFlags The specific `CFLAGS` for a file that must be concatenate with primitive `CFLAGS`.

Note that when using such feature, all files must be list as sources. `example/bar/src`.

If no source pathname is specified, the code generation tools use the entire contents of the `src` directory. The PathName declaration is therefore most useful when selecting a subset of source files residing in the `src` directory or for selecting a source that does not reside in the `src` directory.

4.3.4 Options

The grammar for a compilation option declaration is shown in Listing 4.10.

Listing 4.10: Option grammar specification

```
OptionDeclaration ::=
        ``option'' [TargetName ``.'' ] Name OptionValue

TargetName ::= Name
OptionValue ::=
        Name ( ``|'' Name )* |
        [``-'' | ``+''] INTEGER |
        STRING
```

Where:

Name The name of the option to apply.

TargetName The target where the option Name is valid. Use TargetName to specify options that are relevant only for the given target. (If not specified, the option Name is assumed to be valid for every target.)

OptionValue The decimal, integer (decimal, hexadecimal or octal) or a string value of the option.

The compilation tool chain accepts the following options:

CPPFLAGS "..." The compilation options to pass to the C/C++ pre-processor for pre-processing the component sources.

You can specify top `CPPFLAGS` flags in your `makefile`.

CFLAGS "..." The compilation options to pass to the C/C++ compiler for building the component sources.

`CFLAGS` can be overwritten by a composite component containing this component or by a `CFLAGS` that is defined in the `makefile`.

If you want specific `CFLAGS` to be used to compile generated binding component, you can define the `BC_CFLAGS` flags in your `makefile`.

FIXED_CFLAGS "..." The compilation options to pass to the C compiler for building the component sources.

`FIXED_CFLAGS` is always passed and can not be overwritten.

OVERWRITTEN_CFLAGS "..." The compilation options to pass to the C compiler for building the component sources.

This can overwrite parents `CFLAGS`.

stack DECIMAL The minimum stack size for a component, where `DECIMAL` is a decimal integer. By default, a stack of 128 words is assumed.

memories memory, ... A component only uses a subset of available memories. Memory can be: none, `YRAM`, `EXT24RAM`, `EXT24ROM`, `EXT16RAM`, `EXT16ROM`, `EMB24RAM`, `EMB24ROM`, `EMB16RAM`, `EMB16ROM`. Memory can also be: `none`.

The default is `none`. On composite components, this option is not allowed.

incdir "path;..." The include path to be passed to the C compiler (using `-I`). The path is relative to the component directory storage. Multiple include paths must be delimited by "`;`". On composite components, this option is not allowed.

See subsection 6.1.1 for more information about compilation flags evaluation.

4.3.5 Examples

Following listing shows the `audio.codec.mp3.wrapper` component that extends the `audio.codec` type (`audio/codec/mp3/wrapper/wrapper.adl`). The definition provides a single interface and defines two compilation options.

```
primitive audio.codec.mp3.wrapper extends audio.codec
{
    provides controller.constructor                    as constructor
    option memories EXT24RAM
    option incdir include
}
```

Following listing shows the `audio.periph.dma.dmadriver` component. The audio stack is the dma driver component that uses interface collection and interrupt mechanisms.

```
primitive audio.periph.dma.dmadriver
{
    provides audio.periph.dma.dmadriver.itmanagement as itManagement
    provides exception.handler                 as handler interrupt 30
    requires audio.periph.dma.interruptdma     as itChannel[8] optional
}
```

The purpose of this component is to call right DMA channel callback when DMA interrupt occurs. The `itChannel` interface is declared as a collection with eight entries, the `handler` interface is declared to handle the interrupt line 30.

Following listing shows how to implement an `audio.periph.dma.dmaout` that binds to `dmadriver`, and where `dmachannel` is declared as an attribute that is set by a composite component during static deployment by assigning its value or through component manager during dynamic deployment.

```
primitive audio.periph.dma.dmaout
{
    ...
    attributes t_uint8 channel
    ...
}
```

4.4 Composite components

A composite component is defined by identifying its interfaces, the sub components from which it is constructed, the bindings between those sub components and its properties.

4.4.1 Grammar specifications

Listing 4.11 shows the grammar of a composite component declaration.

Listing 4.11: Composite component grammar specification

```
CompositeDeclaration ::=
        (
            InterfaceDeclaration |
            ContentDeclaration |
            BindingDeclaration |
            PropertyDeclaration
        ) *
        ( OptionDeclaration )*
```

Where:

InterfaceDeclaration Specifies the interfaces of the component. The order in which interfaces are declared is irrelevant; provided (server) and required (client) interface declarations may be interleaved.

ContentDeclaration Specifies the components that are required by the composite component. Because the model is fully recursive, a sub component can be a primitive component or another composite component.

BindingDeclaration Specifies the bindings between the components declared in the `ContentDeclaration` section.

PropertyDeclaration Specifies the properties positioned by the component. A property is a constant that can be read from the host through the component manager.

OptionDeclaration Specifies the compilation options required by the component. Use compilation options to modify the compilation behavior.

A compilation option for a composite component is propagated to all sub components.

4.4.2 Generic Composite

A *generic composite component* allow the parametrization and specialization of composite component declaration either during extending or during definition.

Parametrizing

Listing 4.12 shows the grammar of a composite component declaration allowing to parametrize a composite declaration.

Listing 4.12: Specializing a generic composite

```
CompositeNameDeclaration ::= CompositeName
      [ ``<'' TemplateParameterDeclarations ``>'' ]
      [ ``('' AttributeParameterDeclarations ``)'' ]

TemplateParameterDeclarations ::=
      TemplateParameter ( ``,'' TemplateParameter )*
TemplateParameter ::= ParameterName ``extends'' TypeName

AttributeParameterDeclarations ::=
      AttributeParameter ( ``,'' AttributeParameter )*
AttributeParameter ::=
      PrimitiveType ParameterName [ ``='' AttributeValue ]

ParameterName ::= Name
```

Where:

CompositeName Sepcifies the name of the generic composite template.

ParameterName Specifies the name of a template or attribute parameter.

TypeName Specifies the component type of template parameter.

PrimitiveType Specifies the attribute type of attribute parameter.

AttributeValue Specifies the default attribute value of attribute parameter.

Specializing

Listing 4.13 shows the grammar of a composite component definition allowing to specialize a composite declaration.

Listing 4.13: Specializing a generic composite

```
CompositeDefinition ::= CompositeName
      [ ``<'' ComponentDefinition ( ``,'' ComponentDefinition )* ``>'' ]
      [ ``('' AttributeValue ( ``,'' AttributeValue )* ``)'' ]

ComponentDefinition = PrimitiveName | CompositeDefinition

PrimitiveName ::= FullyQualifedName
CompositeName ::= FullyQualifedName
```

Where:

ComponentDefinition Specified the component definition of a generic composite template parameter.

AttributeValue Specifies the new value of the attribute. Setting this new value is allowed only if the type declared in the generic composite temaplate match with the attribute value.

Like in programming language, specializing parameters don't need to be named and must respect the number and order to the declaration.

4.4.3 Sub components

Listing 4.14 show the grammar of a `ContentDeclaration`.

Listing 4.14: Content grammar specification

```
ContentDeclaration ::=
        ``contains'' ComponentDefinition ``as'' ComponentName [ Priority ]

ComponentName ::= Name

Priority ::= ``urgent'' | ``normal'' | ``background'' | INTEGER
```

Where:

ComponentDefinition The definition of the required component. This component is either a primitive or a specialized composite component definition, also specified using ADL.

ComponentName The local name of the component within this component only.

Priority The priority of the component.

4.4.4 Bindings

The binding declarations define the bindings between interfaces that occur within the component.

Listing 4.15 shows the complete grammar of a binding declaration.

Listing 4.15: Binding grammar specification

```
BindingDeclaration ::=
   ''binds''
       [ ComponentName | ''this'' ] ''.'' ClientInterfaceRawName ''to''
       [ ComponentName | ''this'' ] ''.'' ServerInterfaceRawName
       [
          ''trace'' |
          ''asynchronous'' ''fifo'' ''='' FifoSize
       ]

ClientInterfaceRawName ::= InterfaceName [ ''['' CollectionIndex '']'' ]
ServerInterfaceRawName ::= InterfaceName [ ''['' CollectionIndex '']'' ]

CollectionIndex ::= INTEGER
FifoSize ::= INTEGER
```

For each binding declaration, the names of a pair of interfaces are declared, linked by the "`to`" keyword. NMF always assumes that the first interface named is the client interface and the second interface named is the server interface. The following permutations of interface pairs are permitted.

- The client interface belongs to one sub component and the server interface belongs to another sub component, both of which have been declared as part of the `ContentDeclaration`.
- The client interface belongs to a sub component that has been declared as part of the `ContentDeclaration`, and the server interface is an external interface of the composite component currently being defined.
- The client interface is an external interface of the composite component currently being defined, and the server interface belongs to a sub component that has been declared as part of the `ContentDeclaration`.

Where:

ComponentName.-ClientInterfaceRawName

Specifies the client interface of the binding. This interface either belongs to the sub component identified by `ComponentName`, or, if the keyword `this` is used in place of `ComponentName`, the interface to be bound is an external interface of the composite component currently being defined.

If the required component interface is a collection, $CollectionIndex$ selects a specific interface from the collection. $CollectionIndex$ must be in the range 0 to $CollectionSize - 1$.

ComponentName.-ServerInterfaceRawName

Specifies the server interface of the binding. This interface either belongs to the sub component identified by `ComponentName`, or, if the keyword `this` is used in place of `ComponentName`, the interface to be bound is an external interface of the composite component currently being defined.

If the required component interface is a collection, $CollectionIndex$ selects a specific interface from the collection. $CollectionIndex$ must be in the range 0 to $CollectionSize - 1$.

trace

Specifies that a trace binding component must be added to the binding.

asynchronous fifo

Specifies that an asynchronous communication is required and that an event binding component must be generated to manage the binding.

`FifoSize` specify the maximum number of elements that can be queued.

4.4.5 Composite options

Using the same option syntax defined in subsection 4.3.4 you can define such options for composite component.

LDFLAGS "..."

This option is taken into acccount only for the top composite component and if generating an executable file. It is passed to the linker before the files group.

You can specify top `LDFLAGS` flags in your `makefile`.

LIBLDFLAGS "..."

This option is taken into acccount only for the top composite component and if generating an library file. It is passed to the linker before the files group.

You can specify top `LIBLDFLAGS` flags in your `makefile`.

GROUPLDFLAGS "..." This option is taken into acccount only for the top composite component. It is passed to the linker inside the files group.

You can specify top `GROUPLDFLAGS` flags in your `makefile`.

See subsection 6.1.1 for more information about linker flags evaluation.

4.4.6 Examples

Example 1: Extending types

Following listing shows the `audio.codec.mp3` (`audio/codec/mp3/mp3.adl`) component that extends the `audio.codec` type that was defined in subsection 4.2.5.

```
composite audio.codec.mp3 extends audio.codec
{
    provides controller.constructor     as constructor

    // Sub components
    contains audio.audiolibs.vector     as vector
    contains audio.audiolibs.polysyn    as polysyn
    contains audio.audiolibs.bitstream  as bitstream
    contains audio.audiolibs.mp3hybrid  as mp3hybrid
    contains audio.audiolibs.mp3dequan  as mp3dequan
    contains audio.audiolibs.crc        as crc
    contains audio.audiolibs.audioutils as audioutils
    contains audio.codec.mp3.mp3        as mp3
    contains audio.codec.mp3.wrapper    as wrapper

    // Bind inform
    binds wrapper.outputInform    to this.outputInform

    // Bind with input bit-stream
    binds this.inputConfig        to wrapper.inputConfig
    binds wrapper.inputControl    to this.inputControl
    binds this.inputPort          to wrapper.inputPort
    property inputGranularity = 13824

    // Bind with output PCM stream
    binds wrapper.outputPort      to this.outputPort
    binds this.outputControl      to wrapper.outputControl
    property outputGranularity = 576
}
```

Note that the `inputGranulatity` and `outputGranulatity` properties override the default values given in the component type definition.

Example 2: Using collections

Following listing shows an example that uses collections.

```
composite dma
{
    contains audio.periph.dma.dmadriver as dmadriver
    contains audio.periph.dma.dmain as dmain
    contains audio.periph.dma.dmaout as dmaout

    binds dmadriver.itChannel[0] to dmain.interrupt
    binds dmadriver.itChannel[1] to dmaout.interrupt

    binds dmain.itManagement to dmadriver.itManagement
    binds dmaout.itManagement to dmadriver.itManagement
}
```

Note that the two client interfaces in the `itChannel` collection are each bound to server interfaces belonging to two different sub-components.

The two client interfaces `dmain.itManagement` and `dmaout.itManagement` (which belong to two different sub-components) are both bound to the same server interface (`dmadriver.itManagement`).

Example 3: Using generic composite

Following figure shows an example that using generic component.

A generic composite with one template parameter could be declared like that:

```
type BType { ... }

composite FirstComposite<BType _b>
{
    ...
    contains _b as b
    ...
}
```

Thus, another generic composite that extend it could be declared like that:

```
type AType { ... }

primitive BImpl extends BType { ... }

composite SecondComposite<AType _a> extends FirstComposite<BImpl>
{
    ...
    contains _a as a
    ...
}
```

This declaration could be specialized in definition like that:

```
primitive AImpl extends AType { ... }
```

```
composite ThirdComposite
{
    ...
    contains SecondComposite<Aimpl> as sc
    ...
}
```

A generic composite with one attribute parameter could be declared like that:

```
composite FourthComposite(t_uint32 _t)
{
    ...
    attributes t_uint32 t = _t
    ...
}
```

Thus, another generic composite that extend it and contain a sub-component that extend it also could be declared like that:

```
composite FifthComposite extends FourthComposite(1)
{
    ...
    contains FourthComposite(2) as sc
    ...
}
```

Chapter 5

C/C++ Component Language

The functional content of components are developed using the C or C++ programming language. NMF provides convention and compiler enhancements to the language that aid code development and maintenance.

5.1 C/C++ enhancements

5.1.1 Isolation and interfaces

To enforce the component-interface architecture, there are no variables shared between components nor any undeclared method calls. If communication is required between components, then this can only be achieved by defining appropriate interfaces.

5.1.2 Source files

By convenstion, extension of C source file must be "`.c`" and extension of C++ source file must be "`.cpp`".

5.1.3 Component framework wrapping

For each component, the NMF code generation tools create a file containing information relevant to the compilation of that component. The path of this file is obtained by replacing all occurrences of "`.`" in the component template name to "`/`" and adding an extension of `.nmf`. The developer must add a `#include` directive at the beginning of every C/C++ source code file relating to the component in order to import the contents of this file.

For example, if the component is called `example.foo`, then the generated file with the path `example/foo.nmf` must be included on the first line of each source code file for the component.

```
#include <example/foo.nmf>
/*
 * Source code of component example.foo
 */
...
```

5.1.4 Multiple instances

The main difference between C and C++ is the need to write in C++ an header file that contains the declaration of the component class which include the primitive component context variables, while in C context variables could be declare everywhere in component sources but present some restrictions.

Declaring component context in C

Multi-instance is handle differently according underlying C compiler and deployment time:

PIC/PID compiler with dynamic loading	Mutli-instantiation is handle by duplicating private component data segments and sharing component code segments and shared read-only data segments.
Non PIC/PID compiler with dynamic loading	In this configuration, multi-instantiation is handle by duplicating the binary code in memory.
Static deployement	In this configuration, multi-instantiation is not handle.

Declaring component context in C++

In C++ multi-instance is handle at C++ class level everytime. Thus there is no restriction.

The header file that contains the declaration of the component class which include the primitive component context variables must be declared in a file where the name is obtained by replacing all occurrences of "`.`" in the component template name to "`/`" and addinc `inc` and the small name of the component and adding an extension of `.hpp`.

For example, if the component is called `example.foo` the the `example/foo/inc/foo.hpp` file must be declared and must contains such codes:

```
#ifndef EXAMPLE_FOO_HPP
#define EXAMPLE_FOO_HPP

class example_foo: public example_fooTemplate {
protected:
    // Declare here primitive component context variables
public:
    // Declare virtual method that must be implemented by the class
```

```
};

#endif /*  EXAMPLE_FOO_HPP */
```

NOTE. *The `componentTemplate` class is an abstract class that is defined in the generated `.nmf` file. This generated file also include the `.hpp` file and thus, you don't have to include it in your source code.*

5.1.5 Component compatibility

This table show compatibility between component developped with C or C++.

	Component language	
Project language	C component	C++ component
C project	OK	Non OK
C++ project	OK (but compiled with C++ compiler)	OK

NOTE. *C++ project is switched on through `--cpp` option in tools commmand line.*

5.2 Providing interfaces

A providing interface is one that is declared in the component definition using the keyword `provides`. It is sometimes referred to as a "providing interface" as it provides methods that are called by other components using a client interface.

The C/C++ code in a component must implement all methods of all of that component's server interfaces. Methods that belong to a server interface must be encapsulated using a `METH` macro. Therefore a function named `foo(...)` is actually declared within the source code as:

```
void METH(foo)(...);
```

NOTE. *Singleton provide interfaces in the same way as other components.*

The code generation tools use the `METH` macro to identify the methods that are declared as part of the interface.

5.2.1 Declaring interface methods in C++ header file

In C++, you also have to declare the method in your primitive component class, declared in your `.hpp` file, like that:

```
#ifndef mycomponent_HPP
#define  mycomponent_HPP

class mycomponent: public myComponentTemplate {
    protected:
        t_uint32 myFirstContextVariable;
        void *mySecondContextVariable;
    public:
        virtual void foo(...);
}

#endif /*  mycomponent_HPP */
```

5.2.2 Duplicated methods

Consider a situation in which a component called example.comp provides three server interfaces (using interfaces `A` and `B`) and also requires a client interface (also using interface `B`).

The source code for interface `A` is:

```
interface A {
    void foo();
}
```

The source code for interface `B` is:

```
interface B {
    t_uint32 bar(t_uint32 a);
}
```

The component description is:

```
primitive example.comp
{
    provides A as a1 prefix A1
    provides A as a2 prefix A2
    provides B as b
    requires B as output
}
```

Two implementations of the interface type called A are declared, each with a unique local name (`a1` and `a2`) to distinguish between them. Because the method `foo()` specified by interface type A must be implemented separately by both of these interfaces, there needs to be a way to distinguish between the two versions of this method in the source code. This is achieved by using the ADL keyword prefix to define a unique prefix string for each of the two interfaces. NMF uses this string to prefix the name of the method, creating unique names for the two methods. This means that the two functions to be implemented are called `A1foo()` and `A2foo()` in the source code.

NOTE. *The prefix is used in the source code only. The IDL definitions still use the short name (`foo()`).*

C/C++ source code file implementation

The C/C++ source code of the component consequently contains the methods named in Listing 5.1.

Listing 5.1: Sample code for duplicated methods example

```
#include <example/comp.nmf>
void METH(A1foo)(void) {
    ...
}
void METH(A2foo)(void) {
    ...
}
t_uint32 METH(bar)(t_uint32 a) {
    ...
}
```

C++ header file implementation

The C++ `.hpp` source code of the component consequently contains the methods named in Listing 5.2.

Listing 5.2: Sample code for duplicated methods example in C++ context

```
#ifndef example_comp_HPP
#define  example_comp_HPP

class example_comp: public example_compTemplate {
    protected:
        ...
    public:
        virtual void A1foo(void);
        virtual void A2foo(void);
        virtual t_uint32 bar(t_uint32 a);
}

#endif /*  example_comp_HPP */
```

5.2.3 Server interface collections

A component provides a server interface collection in situations where a number of different components all require access to the same server interface, especially when it might not be known at design time how many components need to access that interface. In order that a server interface can keep track of all the different components that are accessing it, each

client component identifies itself by supplying an index number. The index number falls within a range from 0 to $collectionSize - 1$.

When implementing the methods for the server interface that is to be part of a collection, include the collection index as the final parameter to each method definition. When the method is called by the client, it can use the collection index parameter to identify the calling component.

NOTE. *This parameter needs to be added only to the C/C++ source code of the component, not the IDL definition of the method.*

Implementing a server interface collection has a cost in terms of the size of code. Each index and method declared in the interface adds four assembler instructions to the code size. Although this may not seem excessive, when applied to larger collections, the overhead can quickly multiply. For example, if an interface has three methods used as a collection of eight, NMF adds $3 * 8 * 4 = 96$ assembler instructions to the code.

During binding, the methods that are referenced in the interface descriptor are different for each client, the framework passes this index automatically.

To extend the example given in subsection 5.2.2, redefine provided (server) interface `b` as an interface collection of eight by modifying the configuration file as follows:

```
...
provides B as b[8]
...
```

Use the name `b[i]` (where i is the index number) to reference a specific interface from this collection when defining the bindings of the interfaces in this collection.

Modify the C/C++ source code of the component to:

```
#include <example/comp.nmf>
...
t_uint32 METH(bar)(t_uint32 a, t_uint8 collectionIndex) {
    ...
}
```

The original definition of the function `bar()` had only one parameter, but as it is now part of a collection, an extra parameter has been added in the C source to enable $collectionIndex$ to be passed to the function.

5.2.4 Interrupt handlers

To implement an interrupt handler for an execution engine that calls an interrupt:

1. Declare the interrupt line in the ADL of the component.
2. Add the directive `EE_INTERRUPT` at the beginning of the handler implementation in the source code.

For example, an ADL for an interrupt handler could be as follows:

```
provides exception.handler as cmdit interrupt 14
```

`cmdit.handler` is now the method that is the interrupt handler for interrupt 14. Nothing else needs to be done (either dynamically or statically) to register this handler with the execution engine.

Ensure that the C source for this component contains this code:

```
EE_INTERRUPT void METH(handler)(void) {
    ...
}
```

NOTE. *Interrupts are not de-masked by NMF automatically. De-masking must be performed manually, for example, by implementing a component lifecycle starter and a component lifecycle stopper.*

NOTE. *Interrupt handlers are not yet supported by C++ programming model.*

5.2.5 Unambiguous interface

For enhancing legacy code componentization, the *unambiguous interface* avoid `METH` use for component where user ensure itself that no name conflict could occur (in it'as component itself but also in the big static picture where the component must be declared).

For example, an ADL for an unambiguous interface could be as follows:

```
provides exception.handler as cmdit interrupt 14 unambiguous
```

Ensure that the C source for this component contains this code:

```
EE_INTERRUPT void handler(void) {
    ...
}
```

NOTE. *Unambiguous interface could be combine with unmanaged interface in order to maximize legacy reuse.*

NOTE. *C++ don't support unambiguous interface.*

5.3 Required interfaces

A standard required interface is one that is declared in the component definition by using the keyword requires. It is sometimes referred to as a *requiring interface* as it is an interface that requires appropriate methods to be defined in other components.

If a method called `M(...)` exists in the client interface I, then within a component's source code, the component calls that method with the instruction:

```
I.M(...);
```

The developer does not need to declare the I variable in the source code because it is automatically generated by NMF and declared in the `*.nmf` file, generated by the code generation tools from the `*.adl` file. The generated `*.nmf` file must be imported in the C/C++ source code file using `#include`.

Extending the example first introduced in subsection 5.2.2, the following C/C++ code calls the function called `bar()` in the output interface:

```
#include <example/comp.nmf>

void METH(A1foo)(void) {
    ...
    if(output.bar() == 0) {
        ...
    }
    ...
}
```

5.3.1 Required interface collections

A client interface collection is referenced by using the name of the interface with an array index, for example, `name[i]`. NMF declares a C/C++ variable to allow code to get the size of the collection; this is the name of the interface with `_N` appended, so in this example the variable is called `name_N`.

NOTE. *Only static declarations of collection size are currently permitted, so although `name_N` is declared within the C/C++ code as a variable, its value is fixed at compile time and it should be considered a constant.*

A client interface collection has a cost in terms of the size of data. Each index has an interface descriptor (a word for the context and a word for each method declared in the interface). For example, an interface with three methods used as a collection of eight adds $8 * (1 + 3) = 32$ data words to the component.

During the binding step, to enable binding and calling of different component servers, the referenced interface is different for each index in the interface descriptors array.

To declare a required interface output as an interface collection of 15, the configuration file of the example is modified to include the following line:

```
...
requires B as output[15]
...
```

Use the name `b[i]` (where i is the index number) to reference the correct interface index. Modify the C/C++ source code of the component as follows:

```
#include <example/comp.nmf>
...
t_uint32 METH(bar)(t_uint32 a, t_uint8 collectionIndex) {
    int i;
    for(i = 0; i < output_N; i++)
        output[i].bar(...);
}
```

In the above code fragment, the for loop calls the method `bar(...)` once for each interface in the output interface collection.

5.3.2 Static required interface

A required interface can be declared as *static required interface*. The component directly call the method without specifying the interface, for example, by calling `meth()` instead of `I.meth()`.

This static required interface is called directly without context switching and method indirection, therefore giving better performance during a component call. Such static required interfaces can also be used for legacy code that call existing shared libraries.

The static required can only be bound with singleton components, and it is not possible to distinguish same method name specify in several static required interfaces. Do not require a statically interface that exports the same interface name.

NOTE. *During deployment, binding a static required interface to an interface provide by a singleton component is done exactly in the same way as other bindings, either statically through ADL or dynamically through component manager.*

For example, declare required interface output as a static required interface, the configuration file is modified:

```
...
requires B as output static
...
```

The C source code of the component can be modified, for example:

```
#include <example/comp.nmf>
...
void METH(A1foo)(void) {
    ...
    bar(...);
    ...
}
```

NOTE. *C++ support static require interface and assume that the name was extenal C name by declaring method definition with* `extern ''C''`.

5.3.3 Optional required interface

Required interface contengency could be *optional required interface*. This mean that the component must be able to run either if its optional required interfaces were not bound and thus developer assume to not call optional interface not bound.

Macro `IS_NULL_INTERFACE(itf, meth)` could be used to check if an interface was not bound.

For example, declare required interface output as a optional required interface, the configuration file is modified:

```
...
requires B as output optional
...
```

The C/C++ source code to test an unbound interface look like that:

```
#include <example/comp.nmf>
...
void METH(A1foo)(void) {
    ...
    if(! IS_NULL_INTERFACE(output, bar))
        ouput.bar(...);
    ...
}
```

5.4 Bindings

5.4.1 Static binding optimizations

5.4.2 Interface delegation

Server component

```
interface IRQControler {
    void registerIRQ(
        t_uint32 irqline,
        exception.handler handler);
}
```

```
primitive vic {
    provides IRQControler  as irq
}
```

```
#include <vic.nmf>
Iexception_handler handlers[32];
```

```
void METH(registerIRQ){
    t_uint32 irqline,
    Iexception_handler handler)
{
    ...
    handlers[irqline] = handler;
    ...
}

// In the VIC handler
handler[source].execute();
```

Client component

```
interface exception.handler {
    void execute();
}
```

```
primitive keyboard {
    requires IRQControler  as irq
    provides exception.handler as handler referenced
}
```

```
void run() {
    ...
    irq.registerIRQ(27, handler);
    ...
}
```

5.5 Legacy componentization

Unmanaged interface, unambiguous required interface, static required interface and singleton component has been introduced in order to simplify legacy code componentization.

5.5.1 Library calls

5.5.2 Library implementations

Specifying a unmanaged interface allow to not specifying type already defined in C header file and include them in the IDL. This file is never parsed and thus no binding component could be generated.

```
#include <security/api/type.h>

interface security.api.cryption unmanaged
```

```
{
    t_security_status encrypt(t_key key,
        t_buffer data, t_buffer encryptedData);

    t_security_status decrypt(t_key key,
        t_buffer data, t_buffer encryptedData);
}
```

Specifying a provided interface as unambiguous assume that no interface method name symbol conflict with other symbols, but avoid `METH` macro need in the code.

```
primitive security.lib.crypter {
    provides security.api.cryption as server unambiguous
}
```

Which allow to reuse like that implementation just by including NMF generated file.

```
#include <security/lib/crypter.nmf>

t_security_status encrypt(t_key key,
    t_buffer* data, t_buffer encryptedData) {
    ...
}

t_security_status decrypt(t_key key,
    t_buffer* data, t_buffer encryptedData) {
    ...
}
```

Chapter 6

Build process

The build process of a component is fully handled by the NMF tool chain. No makefile is required.

6.1 Compilation

6.1.1 Compiler flags evaluation

The following rules are used by the NMF tools to determine the compilation flags:

Listing 6.1: Compiler flags evaluation

```
// CPPFLAGS
CPPFLAGS = parent::CPPFLAGS + local::CPPFLAGS
CPPFLAGS += -Iincdir

// CFLAGS
CFLAGS = (local::OVERWRITTEN_CFLAGS == null)?
    ((parent.CFLAGS empty) ?
        local::CFLAGS :
        parent.CFLAGS):
local::OVERWRITTEN_CFLAGS
CFLAGS += local::FIXED_CFLAGS

// LDFLAGS
LDFLAGS = local::LDFLAGS
```

6.1.2 Overwriting binding component CFLAGS

The generated binding components are compiled with `CFLAGS` value that is defined internally according to generated code (by default, this is `-O4`). It is possible to overwrite CFLAGS by defining the `BC_CFLAGS` environment variable: `BC_CFLAGS = -g`

6.2 Errors

The complete list of errors are as follows.

1 BC generator not set, specify -host2mpc, -mpc2host, -asynchronous, ... options.

2 Prefix name for functions array not setted (specify -n <prefixName>).

3 Only gcc and ac compiler and linker are handled (TARGET = targetName).

4 Variable environment envVar must be set.

11 can not compile dynamic component with –pic option.

12 Syntaxic error: ...

13 Lexical error: ...

21 Object fileName does not exists.

22 ADL source fileName not found.

23 Name infered from file <inferedTemplateName> mismatch with specified name <specifiedTemplateName

24 Target named TARGET not recognized.

25 Option named OptionName not recognized.

26 Provide prefix must not be empty.

27 Interface name itfName provide twice in same component.

28 Interface name itfName require twice in same component.

29 Component <templateName> does not require interface itfName.

30 Component <templateName> does not provide interface itfName.

31 Option/Property/Source... declarationName defined twice.

32 Attribute 'pathName' is not valid.

33 Declaring attribute in composite component <templateName> is not allowed.

34 Attribute attributeName is not valid.

35 Option optionName not available with composite component.

36 Prefix perfixValue must start with a letter.

37 Prefix perfixValue must contains only letter or digit.

38 Prefix can not be used in a composite component that contains no sources.

39 Identifier ident of class too large than limit 291.

40 File path not found.

41 Loop in extends stack extendsStack is not allowed.

42 Type name typeName extend twice in same component.

43 Value xxx of class too large than limit 291.

44 Primitive component templateName has no source (src directory is empty).

45 Can not set value to attribute delcared without type.

51 Interface Description Language source fileName not found.

52 Interface Declaration Type source fileName not found.

53 Only inclusion of .idt file allowed.

54 Enumeration enumName not defined in interface is not allowed here.

55 Typedef typedefName not defined in interface is not allowed here.

56 Interface itfType not defined in interface is not allowed here.

57 Structure structName not defined in interface is not allowed here.

58 Constant identifierName not defined in interface is not allowed here.

59 Redeclaration of enumerator enumeratorName.

60 Redeclaration of type typeName.

61 Redeclaration of parameter parameterName.

81 Assigning negative value to unsigned type is not allowed.

82 Assigned this value to string type is not allowed.

83 Assigned string to this type is not allowed.

84 Assigned this value to character type is not allowed.

85 Assigned char to this type is not allowed.

86 Unknown unsigned integer size.

87 Unknown signed integer size.

88 Unknown primitive type.

89 Attribute type 'C-type' not handle by target TARGET.

90 Integer too large than 4095 allowed by type 'C-type'.

91 Integer format not recognized or too long.

32 Constant string xxx too large than limit 291.

101 Component <templateName> requires libc which can not be used with dynamic deployment.

102 Singleton component <templateName> instanciate twice in the same composition.

103 Loop in instantiation stack instantiationStack is not allowed.

104 Sub component componentName already exist in pathName.

105 Sub component componentName does not exist in composite template <templateName>.

106 Can not bind directly composite provide to composite require (this.x − > this.y).

121 Static required interface can only by bound to singleton component.

122 Static interface type itfType can not be required twice by same component.

123 Sub-component localName <templateName> require interface itfName <itfType> not bound.

124 Composite template <templateName> provide interface itfName <itfType> not bound to a sub-component.

125 Composite this.itfname already bound.

126 Interface type mismatch itfType1 != itfType2.

127 Interface itfName already bound.

128 Static interface is not compatible with interface collection.

129 Composite this.itfName not bound.

141 Interface name interfaceName must be of interface type interfaceType.

142 LifeCycle interface interfaceType can not be required by component.

145 LifeCycle interface interfaceType can not be a collection.

149 Number element in collection itfName must be in range [0...255].

150 Interrupt interface must be of interface type rtos.itdispatcher.handler.

151 Interrupt interface can not be a collection.

152 Interrupt line must be in range [2...31].

153 A index must be specify for binding a interface collection templateName.itfName[].

154 Index must be in range 0..255 for interface collection itfName[].

155 A standard interface itfName must not be used with a index.

156 Lifecycle interface can not be provide manually by composite component.

171 Type longer than 32bits not yet allowed for distributed communication.

172 Type 'C-type' of unknown size used in distributed communication.

173 Only method with no return allowed in asynchronouse (distributed) communication.

174 Fifo size out of range [1 .. 256].

255 Error generated by other tools.

CONFIDENTIALITY OBLIGATIONS:

This document contains sensitive information.
Its distribution is subject to the signature of an Non-Disclosure Agreement (NDA).
It is classified "**CONFIDENTIAL**".

At all times you should comply with the following security rules
(Refer to NDA for detailed obligations):

Do not copy or reproduce all or part of this document
Keep this document locked away

Further copies can be provided on a "need to know basis", please contact your local ST-Ericsson sales office.

Please Read Carefully:

Information in this document is provided solely in connection with ST-Ericsson products. ST-Ericsson NV and its subsidiaries ("ST-Ericsson") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST-Ericsson products are sold pursuant to ST-Ericsson's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST-Ericsson products and services described herein, and ST-Ericsson assumes no liability whatsoever relating to the choice, selection or use of the ST-Ericsson products and services described herein. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST-Ericsson for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST-Ericsson'S TERMS AND CONDITIONS OF SALE ST-Ericsson DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST-Ericsson PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY AN AUTHORIZED ST-Ericsson REPRESENTATIVE, ST-Ericsson PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST-Ericsson PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.

Resale of ST-Ericsson products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST-Ericsson for the ST-Ericsson product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST-Ericsson.

ST-Ericsson and the ST-Ericsson logo are trademarks or registered trademarks of ST-Ericsson in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST-Ericsson logo is a registered trademark of ST-Ericsson. All other names are the property of their respective owners.

© 2008 ST-Ericsson - All rights reserved

ST-Ericsson group of companies

Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - India - Italy - Japan - Korea - Malaysia - Mexico - Netherlands - Singapore - Sweden - Switzerland - Taiwan - United Kingdom - United States of America

www.stericsson.com