European Component Oriented Architecture (ECOA®) Collaboration Programme: Preliminary version of the SOFTARC C Language Binding

Dassault Ref No: DGT 2059972-A
Thales DMS Ref No: 69629513-035 --

Issue: 7

Prepared by
Dassault Aviation and Thales DMS

Note: *This specification is preliminary and is subject to further adjustments. Consequently, users are advised to exercise caution when relying on the information herein. No warranties are provided regarding the completeness or accuracy of the information in this preliminary version. The final version of the document will be released to reflect further improvements.*

This Page Intentionally Left Blank

Contents

Figures

Tables

0 Introduction

This Architecture Specification provides the specification for creating ECOA®-based systems. It describes the standardised programming interfaces and data-model that allow a developer to construct an ECOA®-based system. It uses terms defined in the Definitions (Architecture Specification Part 2). The details of the other documents comprising the rest of this Architecture Specification can be found in Section 3.

This document describes the C language binding for ECOA, that is similar to the reference binding ([ECOA C Language Binding]), and compatible with the SOFTARC component framework. SOFTARC is a real-time embedded component framework used by Thales since 2012. This language binding allows SOFTARC components written in C language to be part of the ECOA ecosystem without source code modifications.

This language binding is compatible with the C language standard ANSI X3.159-1989.

This language binding is fully compliant with the generic software interfaces specified in [Architecture Specification Part 4].

This document describes the API identified in ECOA Component Implementation models with the following information:

- APIType = "SOFTARC_C"
- APIVersion = "7.1"

Warning: This document is not exhaustive regarding the Option-specific types and APIs.

This document is structured as follows:

- Section 6 describes the Component to Language Mapping;
- Section 7 describes the method of passing parameters;
- Section 8 describes the Component Context;
- Section 9 describes the basic types that are provided and the types that can be derived from them;
- Section 0 describes the Component Interface;
- Section 11 describes the Container Interface;
- Section 11.8 describes the Container Types;
- Section 14 describes the External Interface;
- Section 13 describes the Default Values;
- Section 14 describes Trigger Instances;
- Section 0 provides a reference C header for the ECOA® library, usable in any C binding implementation;

1 Scope

This Architecture Specification specifies a uniform method for design, development and integration of software systems using a component oriented approach.

2 Warning

This specification represents the output of a research programme. Compliance with this specification shall not in itself relieve any person from any legal obligations imposed upon them. Product development should rely on the BNAE publications of the ECOA standard.

3 Normative References

Architecture Specification Part 2	Dassault Ref No: DGT 2041081-A Thales DMS Ref No: 69398916-035 -- Issue 7 Architecture Specification Part 2 – Definitions
Architecture Specification Part 4	Dassault Ref No: DGT 2041083-A Thales DMS Ref No: 69398918-035 -- Issue 7 Architecture Specification Part 4 – Software Interface
Architecture Specification Part 5	Dassault Ref No: DGT 2041084-A Thales DMS Ref No: 69398919-035 -- Issue 7 Architecture Specification Part 5 – High Level Platform Requirements
Architecture Specification Part 7	Dassault Ref No: DGT 2041086-A Thales DMS Ref No: 69398925-035 -- Issue 7 Architecture Specification Part 7 – Metamodel
ECOA C Language Binding	Dassault Ref No: DGT 2041087-A Thales DMS Ref No: 69398926-035 -- Issue 7 Preliminary version of the ECOA C Language Binding
ANSI X3.159-1989	Programming Languages – C ANSI X3.159-1989

4 Definitions

For the purpose of this standard, the definitions given in Architecture Specification Part 2 apply.

5 Abbreviations

API	Application Programming Interface
ECOA	European Component Oriented Architecture. ECOA® is a registered trademark.
PINFO	Persistent Information
UTC	Coordinated Universal Time
XML	eXtensible Markup Language

6 Component to Language Mapping

6.1 Overview of interfaces

Table 1 lists the Component and Container Interface APIs defined in the "generic" Software Interface document ([Architecture Specification Part 4], Table 1).

The “level” column specifies whether the interface is mandatory (i.e. required in any language binding), or optional.

The "implemented" column specifies whether the interface is implemented in the present language binding.

Table 1 Component and Container InterfacesCategory	**Abstract API Name**	**Level**	**Implemented**
Events API	Event_Send	**MANDATORY**	**YES**
	Event_Received	**MANDATORY**	**YES**
Request Response API	Request_Sync	**MANDATORY**	**YES**
	Request_Async	**MANDATORY**	**YES**
	Request_Received	**MANDATORY**	**YES**
	Response_Received	**MANDATORY***	**YES**
	Response_Actually_Received Response_Not_Received		**NO**
	Response_Send	**MANDATORY**	**YES**
	Request_Cancel	**OPTIONAL**	**YES**
Versioned Data API	Get_Read_Access	**MANDATORY**	**YES**
	Release_Read_Access	**MANDATORY**	**YES**
	Updated	**MANDATORY**	**YES**
	Get_Write_Access	**MANDATORY****	**NO**
	Get_Selected_Write_Access		**YES**
	Cancel_Write_Access	**MANDATORY**	**YES**
	Publish_Write_Access	**MANDATORY**	**YES**
	Is_Initialized	**OPTIONAL**	**YES**
	Release_All_Data_Handles	**OPTIONAL**	**YES**
Properties API	Get_Value	**MANDATORY**	**YES**
Runtime Lifecycle API	Initialize_Received	**MANDATORY**	**YES**
	Start_Received	**MANDATORY**	**YES**

Table 1 Component and Container InterfacesCategory	**Abstract API Name**	**Level**	**Implemented**
	Stop_Received	**MANDATORY**	**YES**
	Shutdown_Received	**MANDATORY**	**YES**
	Reset_Received	**MANDATORY**	**YES**
Supervisor Components	On_State_Change	**OPTIONAL**	**YES [OPTION SUPERVISION]**
	Get_Executable_Status	**OPTIONAL**	
	Executable_Command	**OPTIONAL**	
	Component_State_Command	**OPTIONAL**	
	Get_Component_Status	**OPTIONAL**	
	Get_Variable	**OPTIONAL**	
	Set_Variable	**OPTIONAL**	
Logging and Fault Management Services API	Log_Debug Log_Trace Log_Info Log_Warning Raise_Error Raise_Fatal_Error	**MANDATORY****	**NO**
	Flex_Log Flex_Raise_Fatal_Error		**YES**
	Error_Notification	**OPTIONAL**	**NO**
Time Services API	Get_Relative_Local_Time	**MANDATORY**	**YES**
	Get_UTC_Time	**OPTIONAL**	**NO**
	Get_Absolute_System_Time	**MANDATORY**	**YES**
	Get_Relative_Local_Time_ Resolution	**OPTIONAL**	**NO**
	Get_UTC_Time_Resolution	**OPTIONAL**	**NO**
	Get_Absolute_System_ Time_Resolution	**OPTIONAL**	**NO**
Triggers	Trigger_Set	**MANDATORY**	**YES**
	Trigger_Cancel	**MANDATORY**	**YES**
Persistent Information (PINFO) Management	Read	**MANDATORY**	**YES**
	Write	**OPTIONAL**	**NO**
	Seek	**MANDATORY**	**YES**

Table 1 Component and Container InterfacesCategory	Abstract API Name	Level	Implemented
Context Management	Save_Warm_Start_Context	**OPTIONAL**	**NO**
External Interface	External_Event_Received	**OPTIONAL**	**NO**
External Components	External_Routine	**MANDATORY**	**YES**
	Start_External_Task	**MANDATORY**	**YES**
	Stop_External_Task	**MANDATORY**	**YES**

* it is mandatory to define at least one of the API alternatives in a language binding.

** it is mandatory to define at least one of the API alternatives in a language binding.

In addtion, the present language binding defines the following Component and Container Interface APIs, that are ***not*** defined in the "generic" Software Interface document ([Architecture Specification Part 4]):

- Initialization functions (cf. §9.7.1)
- Comparison functions (cf. §9.7.2)
- Display (cf. §11.3.2.3)

6.2 Overview of files

This section gives an overview of the Component Interface and Container Interface APIs, in terms of the filenames and the overall structure of the files.

With structured languages such as C, the Component Interface will be composed of a set of functions corresponding to each entry-point of the Component Implementation. The declaration of these functions will be accessible in a header file called #component_impl_name#.h. The names of these functions shall begin with the prefix “#component_impl_name#_”.

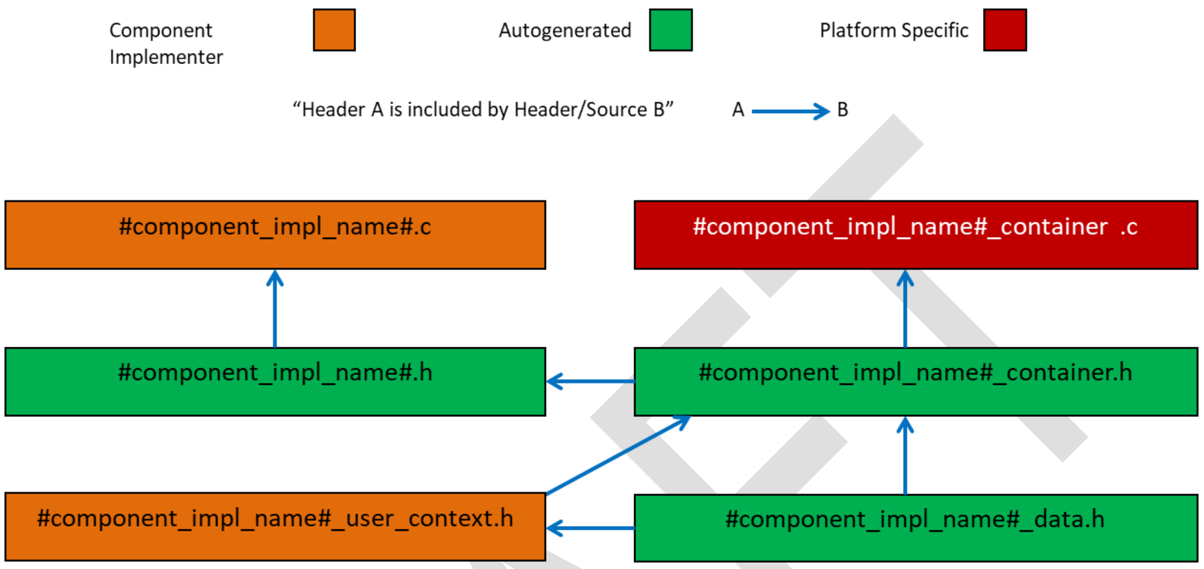
The Container Interface will be composed of a set of functions corresponding to the required operations. The declaration of these functions will be accessible in a header file called #component_impl_name#_container.h. The names of these functions shall begin with the prefix “#component_impl_name#_”.

The Container Types will be composed of the types which the Component Implementation needs in order to declare, use and store various handles. The declaration of these types will be accessible in a header file called #component_impl_name#_data.h. The names of these types shall begin with the prefix “#component_impl_name#_”.

It is important to ensure that the names of these functions and types do not clash within a single executable. One way to achieve this is for each component supplier to define the component implementation name prefixed by a unique identifier. In this way they can manage the uniqueness of their own components, and the mixing of different supplier components within an executable is possible.

A dedicated structure named #component_impl_name#_context, and called Component Context structure in the rest of the document will be generated by the ECOA toolchain in the Component Container header (#component_impl_name#.h) and shall be extended by the Component implementer to contain all the user variables of the Component. This structure will be allocated by the container before Component Instance start-up and passed to the Component Instance in each activation entry-point (i.e. received events, received requests or received asynchronous responses).

Figure 1 shows the relationship between the C files mentioned above, whilst Table 2 shows the filename mappings.



Component Implementer
Autogenerated
Platform Specific
"Header A is included by Header/Source B"
A
B
#component_impl_name#.c
#component_impl_name#_container .c
#component_impl_name#.h
#component_impl_name#_container.h
#component_impl_name#_user_context.h
#component_impl_name#_data.h

Figure 1 C Files Organization

Table 2 Filename Mapping

Filename	Use
#component_impl_name#.h	Component Interface declaration (entry points provided by the component and callable by the container)
#component_impl_name#.c	Component Implementation (implements the component interface)
#component_impl_name#_container.h	Container Interface declaration (functions provided by the container and callable by the component) Component Context type declaration
#component_impl_name#_container.c	Container Implementation: This source (.c) implements the Container Interface. It is out of scope of this document, as it is platform dependent. The Container may actually be a collection of source files depending upon the platform implementation.

Filename	Use
#component_impl_name#_data.h	Container Types declaration (container-level data types usable by the component) These types are related to the Container for a Component Implementation and are declared with the **#component_impl_name#** prefix.
#component_impl_name#_user_context.h	User extensions to Component Context. These types are related to the Component Implementation and are declared with the **#component_impl_name#** prefix.
#library#.h	Types defined from a data type library. Comparison functions.
#library#_initialize.h	Initialisation functions for types defined from a data type library

All of these header files include a reference header file, whose content is given in section §0.

6.3 Component Interface Template

Unspecified by this document.

6.4 Container Interface Template

Unspecified by this document.

6.5 Container Types Template

Unspecified by this document.

6.6 User Component Context Template

Unspecified by this document.

6.7 Guards

In C, all of the declarations within header files (files with extension .h) shall be surrounded within the following block to make the code compatible with C++, and to avoid multiple inclusions:

```
#if !defined(#macro_protection_name#_H)
#define #macro_protection_name#_H

#if defined(_cplusplus)
extern "C" {
#endif /* _cplusplus */

/* all the declarations shall come here */
#if defined(_cplusplus)
}
#endif /* _cplusplus */

#endif  /* #macro_protection_name#_H */
```

Where #macro_protection_name# is the name of the header file in capital letters and without the .h extension.

7 Parameters

This section describes the manner in which parameters are passed in C:

Input parameters will be passed as pointers to a const; output parameters defined with a complex type will be passed as pointers.

Table 3 Method of Passing Parameters

	Input parameter	**Output parameter**
Simple type	Pointer to const	Pointer
Complex type	Pointer to const	Pointer

Within the API bindings, parameters will be passed as constant if the behaviour of the specific API warrants it. This will override the normal conventions defined above.

8 Component Context

In the C language, the Component Context is a structure which holds both the user local data (called “User Component Context”) and infrastructure-level technical data (which is implementation dependant). The warm start context feature may be optionally selected in the Component Implementation model using option 'hasWarmStartContext'. The presence or absence of declarations of corresponding fields in Component code must be in accordance with selections made in the Component Implementation model. The structure is defined in the Container Interface.

Any language type can be used within the contexts (including ECOA ones).

The following shows the C syntax for the Component Context:

```
/* @file “#component_impl_name#_container.h”
 * Container Interface header for Component #component_impl_name#
 * Generated automatically from specification; do not modify here
 */

/* Container Types */
#include “#component_impl_name#_data.h”

/* User Context */
#include “#component_impl_name#_user_context.h”

/* Incomplete definition of the technical (platform-dependent) part of the */
/* context (it will be defined privately by the container) */
struct #component_impl_name#_platform_hook;

/* Component Context structure declaration */
typedef struct
{
  /* User context */
  #component_impl_name#_user_context user;

  /* Pointer to technical context */
  struct #component_impl_name#_tech_context *tech;

} #component_impl_name#_context;
```

8.1 User Component Context

The following shows the C syntax for the Component User Context (including an example data item; myCounter):

```
/* @file #component_impl_name#_user_context.h
 * This is an example of a user defined User Component context
 */

```

```
/* Container Types */
#include "#component_impl_name#_data.h"

/* User Component Context structure example */
typedef struct
{
   /* declare the User Component Context "local" data here */
   int myCounter;
} #component_impl_name#_user_context;
```

Data declared within the Component User Context can be of any type.

The following example illustrates the usage of the Component context in the entry-point corresponding to an event-received:

```
/* @file "#component_impl_name#.c"
 * Generic operation implementation example
 */

void #component_impl_name#_#operation_name#_received
   (#component_impl_name#_context* context)
{
   /* To be implemented by the component */

   /*
    * …
    * increments a local user defined counter:
    */
   context->user.myCounter++;
}
```

The user extensions to Component Context need to be known by the container in order to allocate the required memory area. This means that the component supplier is required to provide the associated header file. If the supplier does not want to divulge the original contents of the header file, then:

- It may be replaced by an array with a size equivalent to the original data; or
- Memory management may be dealt with internally to the code, using memory allocation functions[1]
- The size of the Component User Context may be declared in the bin-desc file related to the Component.

To extend the Component Context structure, the component implementer shall define the User Component Context structure, named #component_impl_name#_user_context, in a header file called

1 The current ECOA **Erreur ! Source du renvoi introuvable.** does not specify any memory allocation function. So, this c ase may lead to non-portable code.

#component_impl_name#_user_context.h. All the private data of the Component Implementation shall be added as members of this structure, and will be accessible within the “user” field of the Component Context.

The Component Context structure will be passed by the Container to the Component as the first parameter for each operation (i.e. received events, received requests or received asynchronous responses). The Component Context defines the instance of the Component being invoked by the operation. This structure shall be passed by the Component to all Container Interface API functions it can call.

8.2 Warm Start Component Context

The "warm start context" is not supported by the SOFTARC C language binding.

If a component implementation has a warm start component context and uses this language binding, a generation error shall be raised.

9 Types

This section describes the convention for creating type libraries, and how the ECOA basic types and derived types are represented in C.

9.1 Libraries

The type definitions are contained within libraries: all types for specific library defined in `#library#.types.xml` shall be placed in a file called `#library#.h`

The complete name of the declaration of a variable name and type name will be computed by prefixing these names with the name of the library. In the C language, this naming rule will be used for each variable or type declaration to create the complete variable name, reflecting the library from which it is defined.

Below is an example of a simple type being defined within a library in C.

```
/*
 * @file #library#.h
 * Data-type declaration file
 * Generated automatically from specification; do not modify here
 */

typedef #basic_type_name#
  #library#_#simple_type_name#;
```

9.2 Basic Types

The basic types, shown in Table 4, shall be located in the “ECOA” library and hence in ECOA.h which shall also contain definitions of the pre-defined constants, e.g. that define constants to represent the true and false values of the basic Boolean type, that are shown in **Erreur ! Source du renvoi introuvable.**.

Table 4 C Basic Type Mapping

ECOA Basic Type	C type	C constants
`ECOA.boolean8`	`SARC_boolean8`	`SARC_TRUE, SARC_FALSE`
`ECOA.int8`	`SARC_int8`	
`ECOA.char8`	`SARC_char8`	
`ECOA.byte`	`SARC_Byte`	
`ECOA.int16`	`SARC_int16`	
`ECOA.int32`	`SARC_int32`	
`ECOA.int64`	`SARC_int64`	
`ECOA.uint8`	`SARC_uint8`	
`ECOA.uint16`	`SARC_uint16`	
`ECOA.uint32`	`SARC_uint32`	
`ECOA.uint64`	`SARC_uint64`	

ECOA Basic Type	C type	C constants
ECOA.float32	**SARC_float32**	
ECOA.double64	**SARC_double64**	

Note: The macros defined in standard header files "stdint.h" and "float.h" indicate the limits of these types.

9.3 Derived Types

Note that as namespaces are not supported in the C language, the actual name of a type (known as the complete type and referred to here by prefixing `complete_`) will be computed by adding as a prefix the name of the library in which it is included.

9.3.1 Simple Types

The syntax for defining a Simple Type #simple_type_name# refined from a Basic Type #basic_type_name# in C is defined below.

```
typedef #basic_type_name# #complete_simple_type_name#;
```

If the optional #minRange# or #maxRange# fields are set, the previous type definition must be followed by the minRange or maxRange constant declarations as follows:

```
#define #complete_simple_type_name#_minRange (#minrange_value#)
#define #complete_simple_type_name#_maxRange (#maxrange_value#)
```

9.3.2 Enumerations

The C syntax for defining an enumerated type named #enum_type_name#, with a set of labels named from #enum_type_name#_#enum_value_name_1# to #enum_type_name#_#enum_value_name_n# and a set of optional values of the labels named #enum_value_value_1# … #enum_value_value_n# is defined below.

```
typedef #basic_type_name# #complete_enum_type_name#;

#define #complete_enum_type_name#_#enum_value_name_1#   (#enum_value_value_1#)
#define #complete_enum_type_name#_#enum_value_name_2#   (#enum_value_value_2#)
#define #complete_enum_type_name#_#enum_value_name_3#   (#enum_value_value_3#)
/*…*/
#define #complete_enum_type_name#_#enum_value_name_n#   (#enum_value_value_n#)
```

Where:

`#basic_type_name#` is the other type on which this enumeration is based.

`#complete_enum_type_name#` is computed by prefixing the name of the type with the namespaces and using '_' as separator.

`#enum_value_value_X#` is the optional value of the label. If not set, this value is computed from the previous label value, by adding 1 (or set to 0 if it is the first label of the enumeration).

9.3.3 Records

For a record type named #record_type_name# with a set of fields named #field_name1# to #field_namen# of given types #data_type_1# to #data_type_n#, the syntax is given below.

The order of fields in the struct shall follow the order of fields used in the XML definition.

```
typedef struct
{
   #data_type_1# #field_name1#;
   #data_type_2# #field_name2#;
   /*…*/
   #data_type_n# #field_namen#;
}  #complete_record_type_name#;
```

9.3.4 Variant Records

For a Variant Record named #variant_record_type_name# containing a set of fields (named #field_name1# to #field_namen#) of given types #data_type_1# to #data_type_n# and other optional fields (named #optional_field_name1# to #optional_field_namen#) of type (#optional_type_name1# to #optional_type_namen#) with selector #selector_name#, the syntax is given below.

The order of fields in the struct shall follow the order of fields used in the XML definition.

```
typedef struct{

    #complete_selector_type_name# #selector_name#;

    #data_type_1# #field_name1#; /* for each <field> element */
    #data_type_2# #field_name2#;
    /*…*/
    #data_type_n# #field_namen#;

    union  {
       #optional_type_name1# #optional_field_name1#; /* for each <union>
         element */
       #optional_type_name2# #optional_field_name2#;
       /*…*/
       #optional_type_namen# #optional_field_namen#;
    } u_#selector_name#;

} #complete_variant_record_type_name#;
```

9.3.5 Fixed Arrays

The C syntax for a fixed array named #array_type_name# of maximum size #max_number# and element type of #data_type_name# is given below.

A macro called #complete_array_type_name#_MAXSIZE will be defined to specify the size of the array.

```
#define #complete_array_type_name#_MAXSIZE #max_number#
typedef struct {
   #data_type_name# values[#complete_array_type_name#_MAXSIZE];
} #complete_array_type_name#;
```

9.3.6 Variable Arrays

The C syntax for a variable array (named #var_array_type_name#) with maximum size #max_number#, elements with type #data_type_name# and a current size of current_size is given below.

```
#define #complete_var_array_type_name#_MAXSIZE #max_number#
typedef struct {
   SARC_uint32 size;
   #data_type_name# values[#complete_var_array_type_name#_MAXSIZE];
} #complete_var_array_type_name#;
```

9.4 Predefined Types

9.4.1 Function execution return status

In C `ECOA.return_status` translates to `SARC_Ecode`, with the enumerated values shown below:

```
typedef enum
{
  /** Used when function behaved as expected */
  SARC_SUCCESS = 0,
  /** Most of the time, abnormal behaviour results in a FAILURE */
  SARC_FAILURE = 1,
  /** When a function returns because its execution time slot has expired */
  SARC_TIMEOUT = 7,
  /** Alias for SARC_SUCCESS - deprecated */
  SARC_OK = SARC_SUCCESS,
  /** Alias for SARC_FAILURE - deprecated */
  SARC_KO = SARC_FAILURE,
  SARC_INVALID_IN_PARAMETER = 2,
  SARC_INVALID_OUT_PARAMETER = 3,
  SARC_INVALID_DATA = 4
} SARC_Ecode;
```

The mapping of this type `SARC_Ecode` to the abstract type `ECOA.return_status` defined in [Architecture Specification Part 4] is given by the following table:

Value in SARC_Ecode	Value in ECOA.return_status	Comment

SARC_SUCCESS SARC_OK (1)	ECOA:OK	No error has occurred
SARC_FAILURE SARC_KO (2)	ECOA:FAILURE	Generic default return status code for non-nominal execution
SARC_TIMEOUT	ECOA:NO_RESPONSE (3)	No response was received for a request (timeout reached)
SARC_INVALID_IN_PARAMETER	ECOA:INVALID_IN_PARAMETER (3)	An invalid IN parameter has been used
SARC_INVALID_OUT_PARAMETE R	ECOA:INVALID_OUT_PARAMETER (3)	An invalid OUT parameter has been used
SARC_INVALID_DATA	ECOA:INVALID_PARAMETER (3)	An invalid DATA operation value has been used

The reverse mapping is given by the following table:

Value in ECOA.return_status	**Value in SARC_Ecode**	**Comment**
ECOA:OK	SARC_OK	No error has occurred
ECOA:FAILURE	SARC_FAILURE	Generic default return status code for non-nominal execution
ECOA:INVALID_HANDLE	SARC_FAILURE	An invalid handle has been used
ECOA:DATA_NOT_INITIALIZED	SARC_FAILURE	The data has never been written
ECOA:NO_DATA	SARC_FAILURE	The call is not able to provide any data
ECOA:INVALID_IDENTIFIER	SARC_FAILURE	An invalid ID has been used.
ECOA:NO_RESPONSE	SARC_TIMEOUT (if timeout) SARC_FAILURE (otherwise)	No response was received for a request
ECOA:OPERATION_ALREADY_ PENDING	SARC_FAILURE	The requested operation is already being processed
ECOA:CLOCK_UNSYNCHRONIZ ED	SARC_FAILURE	The clock is not synchronised
ECOA:RESOURCE_NOT_AVAIL ABLE	SARC_FAILURE	Insufficient resource is available to perform the operation.
ECOA:OPERATION_NOT_AVAIL ABLE	SARC_FAILURE	The requested operation is not available.
ECOA:INVALID_PARAMETER ECOA:INVALID_IN_PARAMETER ECOA:INVALID_OUT_PARAMET ER	SARC_INVALID_DATA SARC_INVALID_IN_PARAMETER SARC_INVALID_OUT_PARAMET ER	An invalid parameter has been used An invalid IN parameter has been used An invalid OUT parameter has been used (2)

(1) SARC_OK and SARC_KO are aliases for SARC_SUCCESS and SARC_FAILURE, defined only for compatibility reasons. The use of SARC_SUCCESS and SARC_FAILURE is encouraged.
(2) ECOA:INVALID_PARAMETER is equivalent to SARC_INVALID_DATA in the case of a "data" operation only; else, SARC_INVALID_IN_PARAMETER (for "event" and "request-response" operations) or SARC_INVALID_OUT_PARAMETER (for "request-response" operations) shall be used.
(3) or ECOA:FAILURE if this code is not supported by the targeted binding.

Nota

- The default value to use in ECOA.return_status when nothing else is available is ECOA:FAILURE.
- The default value to use in SARC_Ecode when nothing else is available is SARC_FAILURE.

9.4.2 Component and executable identifiers

In this language binding, components and executables are not identified by a specific type but using the SARC_int32 predefined type.

9.4.3 Write access mode

`ECOA:write_access_mode` translates to `SARC_DataValue`, with the enumerated values shown below:

```
/**
 * This enumerated type describes the possible ways a data
 * handle can be initialized.
 */
typedef enum
{
    /**
     * The memory area pointed by the handle has no specific value
     * Use this by default in order to have better performances.
     */
  SARC_DATA_NO_VALUE,

    /**
     * The memory area pointed by the handle is filled with the
     * current data value. This mode has lower performances, but
     * allows access the current data version before modifying it.
     */
  SARC_DATA_CURRENT_VERSION
} SARC_DataValue;
```

9.4.4 Time management

For time management interfaces, the present binding uses the abstract type `ECOA.nano_time` defined in [Architecture Specification Part 4], defined as SARC_int64.

Abstract type	Type in binding
ECOA.hr_time	*(not defined)*
ECOA.global_time	*(not defined)*
ECOA.duration_time	*(not defined)*
ECOA.nano_time	SARC_int64

9.4.5 Logs

In this language binding, no specific type is defined for logs.

9.4.6 Error management

In this language binding, no specific type is defined for error management.

9.4.7 Pinfo management

In this language binding, the data type ECOA.seek_whence_type is named SARC_RomOrigin.

The C syntax for this type is:

```
/** Reference position used when moving the reading head */
typedef enum
{
 /** Beginning of ROM */
  SARC_ROM_ORIGIN_START,
  /** Current position */
  SARC_ROM_ORIGIN_CURRENT,
  /** End of ROM */
  SARC_ROM_ORIGIN_END
} SARC_RomOrigin;
```

9.4.8 Lifecycle management

- **ECOA.component_state**

The C syntax for this type is:

```
typedef SARC_uint32 SARC_LifeCycleState;

/** UNAVAILABLE Instance state */
#define SARC_LIFE_CYCLE_STATE_UNAVAILABLE 0x00
/** IDLE Instance state */
#define SARC_LIFE_CYCLE_STATE_IDLE 0x01
/** READY Instance state */
#define SARC_LIFE_CYCLE_STATE_READY 0x02
```

```
/** RUNNING Instance state */
#define SARC_LIFE_CYCLE_STATE_RUNNING 0x03
```

- **ECOA.component_command**

The C syntax for this type is:

```
typedef SARC_uint32 SARC_LifeCycleShift;

/** INITIALIZE transition */
#define SARC_LIFE_CYCLE_SHIFT_INITIALIZE 0x02
/** START transition */
#define SARC_LIFE_CYCLE_SHIFT_START 0x03
/** RESET transition */
#define SARC_LIFE_CYCLE_SHIFT_RESET 0x04
/** STOP transition */
#define SARC_LIFE_CYCLE_SHIFT_STOP 0x05
/** SHUTDOWN transition */
#define SARC_LIFE_CYCLE_SHIFT_SHUTDOWN 0x06
/** KILL transition */
#define SARC_LIFE_CYCLE_SHIFT_KILL 0x07
```

- **ECOA.executable_state**

The C syntax for this type is:

```
typedef SARC_uint32 SARC_ExecutableStates;

/** Tag for unavailable executable status */
#define SARC_STATUS_EXECUTABLE_NULL 0
/** Tag for running executable status */
#define SARC_STATUS_EXECUTABLE_LAUNCHED 1
```

- **ECOA.executable_command**

The C syntax for this type is:

```
typedef SARC_uint32 SARC_ExecutablesCommands;

#define SARC_PANEL_COMMAND_LAUNCH 1
#define SARC_PANEL_COMMAND_KILL 2
```

9.5 Constants

The syntax for the declaration of a Constant called “`#constant_name#`” in C is shown below. Note that the `#type_name#` is not used in the C binding. In addition, namespaces are not supported in the C language, so the name of the constant (known as the complete name and referred to here by prefixing `complete_`) will be computed by adding as a prefix the name of the library in which it is included.

```
#define #complete_constant_name# (#constant_value#)
```

where #constant_value# is either an integer or floating point value described by the XML description.

9.6 Predefined constants

The constant *#component_impl_name#_#operation_name#_MAX_CONCURRENT_REQUESTS* is implemented in the present language binding with the following C language syntax:

```
#define #component_impl_name#_#operation_name#_SERVICE_MAXDEFERRED
#max_concurrent_requests#
```

9.7 Functions defined on types

9.7.1 Initialization functions

For each ECOA library, a file called #library#_initialize.h contains initialization functions. Each type defined in the library has an initialization function conforming to the following declaration:

```
void #library#_#type_name#_initialize (#library#_#type_name# *value);
```

This method initializes 'value' with a default value, which is defined as:

- For boolean types, the default value is false.
- For scalar types, the default value is 0.
- For enumerated types, the default value is the first value.
- For record types, the default value is made of the default value of each field.
- For variant record types, the default value is made of the default value of each field, and the selector has the default value of its type.
- For fixed array types, the default value is made of the default value of each element.
- For array types, the default value is the empty array.

9.7.2 Comparison functions

For each ECOA library, the same file #library#.h contains comparison functions. Each type defined in the library, except scalar types (enum and simple types), has an comparison function conforming to the following declaration:

```
SARC_boolean8 #library#_#type_name#_equals (
  const #library#_#type_name# *v1, *v2);
```

This method returns true if the types have an equal value and false otherwise. Note that contrary to a memcmp() function call, only the significant bytes are compared. Alignment gaps, unused memory in array or variantrecord types is not taken onto account.

10 Component Interface

10.1 Operations

This section contains details of the operations that comprise the component API i.e. the operations that can invoked by the container on a component.

10.1.1 Request-Response

10.1.1.1 Request Received

The following is the C syntax for invoking a request received by a component instance when a response is required, where #component_impl_name# is the name of the component implementation receiving the request and #operation_name# is the operation name. The same syntax is applicable for both synchronous and asynchronous request-response operations.

10.1.1.1.1 Request Received when immediate=false

```
void #component_impl_name#_#operation_name#_SERVICE_provide
   (#component_impl_name#_context* context,
    SARC_uint32 request_id,
    const #request_parameters#);
```

10.1.1.1.2 Request Received when immediate=true

```
void #component_impl_name#_#operation_name#_SERVICE_provide
   (#component_impl_name#_context* context,
    const #request_parameters#,
    #response_parameters#
);
```

10.1.1.2 End of an asynchronous Request

10.1.1.2.1 Response Received

The following is the C syntax for an operation used by the container to send the response to an asynchronous request response operation to the component instance that originally issued the request, where #component_impl_name# is the name of the component implementation receiving the response and #operation_name# is the operation name. (The reply to a synchronous request response is provided by the return of the original request).

```
void #component_impl_name#_#operation_name#_Callback
   (#component_impl_name#_context* context,
    SARC_uint32 request_id,
    const #response_parameters#);
```

The "`#response_parameters#`" are the "out" parameters of the request-response operation, but are treated as inputs to the function and passed as "const" parameters, so they are not modified by the component.

The following is the C syntax for an operation used by the container to signal a timeout (i.e. no response received after the given time) on the response to an asynchronous request response operation:

```
void #component_impl_name#_#operation_name#_Timeout
   (#component_impl_name#_context* context,
    SARC_uint32 request_id);
```

10.1.1.2.2 Optional Alternative

10.1.1.2.2.1 Response_Actually_Received

Not available in this binding.

10.1.1.2.2.2 Response_Not_Received

Not available in this binding.

10.1.2 Versioned Data Updated

The following is the C syntax that is used by the container to inform a component instance that reads an item of versioned data that new data has been written.

```
void #component_impl_name#_sarc_notify_#operation_name#_EVENT_receive
   (#component_impl_name#_context* context);
```

10.1.3 Event Received

The following is the C syntax for an event received by a component instance.

```
void #component_impl_name#_#operation_name#_EVENT_receive
   (#component_impl_name#_context* context,
    const #event_parameters#);
```

10.2 Component Lifecycle

The following operations are applicable to application, trigger and dynamic-trigger component instances.

10.2.1 Initialize_Received

The C syntax for an operation to initialise a component instance is:

```
void #component_impl_name#_initialize
   (#component_impl_name#_context* context);
```

10.2.2 Start_Received

The C syntax for an operation to start a component instance is:

```
void #component_impl_name#_start
```

```
   (#component_impl_name#_context* context);
```

10.2.3 Stop_Received

The C syntax for an operation to stop a component instance is:

```
void #component_impl_name#_stop
   (#component_impl_name#_context* context);
```

10.2.4 Shutdown_Received

The C syntax for an operation to shutdown a component instance is:

```
void #component_impl_name#_shutdown
   (#component_impl_name#_context* context);
```

10.2.5 Reset_Received

This entry point is only available if the component implementation model has set the option ***hasReset***.

The C syntax for an operation to (functionally) reset a component instance is:

```
void #component_impl_name#_reset
   (#component_impl_name#_context* context);
```

10.3 Supervisor components

The following C types and methods are applicable to application component instances of kind SUPERVISOR, to be informed of lifecycle state changes of component instances of the application.

```
void #component_impl_name#_onStateChange
   (#component_impl_name#_context* context,
        SARC_int32 componentInstanceId,
        SARC_int32 state,
        SARC_int32 oldState);
```

state represents the new state of the component instance identified with the componentId, among the following set:

- SARC_LIFE_CYCLE_STATE_IDLE;
- SARC_LIFE_CYCLE_STATE_READY;
- SARC_LIFE_CYCLE_STATE_RUNNING;
- SARC_LIFE_CYCLE_STATE_UNAVAILABLE.

oldState represents the old state of the component instance identified with the componentId, among the following set:

- SARC_LIFE_CYCLE_STATE_IDLE;
- SARC_LIFE_CYCLE_STATE_READY;
- SARC_LIFE_CYCLE_STATE_RUNNING;
- SARC_LIFE_CYCLE_STATE_UNAVAILABLE.

10.4 Error notification for fault handler components

Not applicable to this binding.

11 Container Interface

11.1 Operations

11.1.1 Request Response

11.1.1.1 Synchronous Request

The C syntax for a component instance to perform a synchronous request response operation is:

```
SARC_Ecode
  #component_impl_name#_#operation_name#_SERVICE_call
   (#component_impl_name#_context* context,
    const #request_parameters#,
    #response_parameters#);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_INVALID_IN_PARAMETER: invalid value in service input parameters;
- SARC_INVALID_OUT_PARAMETER: invalid value in service output parameters;
- SARC_TIMEOUT: called function timed out;
- SARC_KO: failed service call (invalid instance or there is actually no server).

11.1.1.2 Asynchronous Request

The C syntax for a component instance to perform an asynchronous request response operation is:

```
SARC_Ecode
  #component_impl_name#_#operation_name#_ASYNC_SERVICE_call
   (#component_impl_name#_context* context,
    SARC_uint32* request_id,
    const #request_parameters#);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_INVALID_IN_PARAMETER: invalid value in service input parameters;
- SARC_KO: failed service call (invalid instance or there is actually no server).

11.1.1.3 Response Send

The C syntax, applicable to both synchronous and asynchronous request response operations, for sending a reply is:

```
SARC_Ecode
  #component_impl_name#_#operation_name#_SERVICE_reply
   (#component_impl_name#_context* context,
    const SARC_uint32 request_id,
    const #response_parameters#);
```

The “#response_parameters#” are the “out” parameters of the request-response operation, but are treated as inputs to the function and passed as “const” parameters, so they are not modified by the container. The request_id parameter is that which is passed in during the invocation of the request received operation.

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_INVALID_OUT_PARAMETER: invalid value in service output parameters;
- SARC_KO: failed service call (invalid instance or there is actually no server).

The following constant allows the component to be aware of the maximum number of requests defined in the component's model by attribute 'maxConcurrentRequests'. This information is useful to allocate memory for storage of the interna data associated to the handling ot these pending requests:

```
#define #component_impl_name#_#operation_name#_SERVICE_MAXDEFERRED
#max_concurrent_requests#
```

11.1.1.4 Request Cancel

The following function allows the component to cancel the handling of a pending request, i.e. notify the infrastructure and the caller that no response will be sent for this request. If a timeout value is defined, the timeout expiration is anticipated, i.e. the caller does not wait until timeout expiration and receives a FAILURE return code.

```
SARC_Ecode
  #component_impl_name#_#operation_name#_SERVICE_cancel
   (#component_impl_name#_context* context,
    const SARC_uint32 request_id);
```

11.1.2 Versioned Data

This section contains the C syntax for versioned data operations, which allow a component instance to

- Get (request) Read Access
- Release Read Access
- Get (request) Write Access
- Cancel Write Access (without writing new data)
- Publish (write) new data (automatically releases write access)

- Note: the definition of versioned data handles involved in all #operation_name# is done in the Container Types header file, as specified in Section 12.1.1.

11.1.2.1 Get Read Access

```
#include “#component_impl_name#_data.h”
```

```
SARC_Ecode
  #component_impl_name#_#operation_name#_DATA_get_reader
   (#component_impl_name#_context* context,
    t_#operation_name#_handle* data_handle);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call (invalid instance; or value has not yet been published; or maximum reader versions count reached for the component).

11.1.2.2 Release Read Access

```
#include “#component_impl_name#_data.h”

SARC_Ecode
  #component_impl_name#_#operation_name#_DATA_release
   (#component_impl_name#_context* context,
    t_#operation_name#_handle* data_handle);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call (invalid instance; or value has no concrete existence; or accessor is corrupted).

11.1.2.3 Get Write Access

Not available in this binding.

Use "Get Selected Write Access" defined in next section.

11.1.2.4 Get Selected Write Access

```
#include “#component_impl_name#_data.h”

SARC_Ecode
  #component_impl_name#_#operation_name#_DATA_get_writer
   (#component_impl_name#_context* context,
    t_#operation_name#_handle* data_handle,
    SARC_DataValue init);
```

The parameter 'init' specifies if the memory area pointed by 'handle' should be initialized by the infrastructure or not:

- SARC_DATA_NO_VALUE: do not initialize the area. It may contain any value. Use this to optimize execution time when the application code will fully initialize the area anyway.
- SARC_DATA_CURRENT_VERSION: initialize the area with the current version of the data. If it does not exist, or if the DataWritten operation has the 'writeOnly' atttribute set in the component's model, then SARC_Ecode will be SARC_KO. Use this when the application code only updates the area

before publishing the new value. This case corresponds to the Get Overwrite Access operation from [Architecture Specification Part 4].

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call (invalid instance; or all DataVR slots are already in use; or no value have been published yet and initialization is requested).

11.1.2.5 Cancel Write Access

```
#include "#component_impl_name#_data.h"

SARC_Ecode
  #component_impl_name#_#operation_name#_DATA_cancel
   (#component_impl_name#_context* context,
    t_#operation_name#_handle* data_handle);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call (invalid instance or accessor is corrupted).

11.1.2.6 Publish Write Access

```
#include "#component_impl_name#_data.h"

SARC_Ecode
   #component_impl_name#_#operation_name#_DATA_publish
   (#component_impl_name#_context* context,
    t_#operation_name#_handle* data_handle);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_INVALID_DATA: invalid value in to-be-published data;
- SARC_KO: failed operation call (invalid instance or accessor is corrupted).

11.1.2.7 Is Initialized

This function allows the component to know if a data has a value or not, i.e. if it has been initialized, either by a default value in the assembly, or by a write operation.

```
#include "#component_impl_name#_data.h"

SARC_boolean8
   #component_impl_name#_#operation_name#_is_initialized
   (#component_impl_name#_context* context);
```

11.1.2.8 Release All Data Handles

This function allows to release all data handles (read and write) obtained by the component. It can be used to simplify by ensuring at a given point in code that no handle is kept by the component.

```
#include "#component_impl_name#_data.h"

void
   #component_impl_name#_#operation_name#_release_all_data_handles
   (#component_impl_name#_context* context);
```

11.1.3 Events

11.1.3.1 Send

The C syntax for a component instance to perform an event send operation is:

```
SARC_Ecode #component_impl_name#_#operation_name#_EVENT_send
   (#component_impl_name#_context* context,
    const #event_parameters#);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: event sent properly;
- SARC_KO: event not sent, because of invalid context pointer.

11.2 Properties

This section describes the syntax for the Get_Value operation to request the component properties whose values are fulfilled by the Infrastructure based on elements described in the component implementation XML file.

11.2.1 Get Value

The syntax for Get_Value is shown below, where

- `#property_name#` is the name of the property used in the component definition,
- `#property_type_name#` is the name of the data-type of the property.

```
SARC_Ecode #component_impl_name#_get_attribute_#property_name#
   (#component_impl_name#_context* context,
    #property_type_name#* value);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call.

11.2.2 Expressing Property Values

Not applicable to the C Binding.

11.2.3 Example of Defining and Using Properties

Not applicable to the C Binding.

11.3 Logging and Fault Management

This section describes the C syntax for the logging and fault management operations provided by the container.
This language binding uses "Alternative 2 : using flex interfaces" defined in [Architecture Specification Part 4].

In the following subsections, the interpretation of 'format' and the supplemental parameters ("…") is delegated to the function 'vsnprintf' (from the C standard library).

11.3.1 Alternative 1: using fixed interfaces

Not available in this binding.

11.3.2 Alternative 2: using flex interfaces

11.3.2.1 Flex_Log

```
void #component_impl_name#_trace
   (#component_impl_name#_context* context, SARC_TraceLevel level,
    const char *format, ...);
```

The formatted string is truncated to 1024 characters, including a teminating null byte.

The type SARC_TraceLevel is defined as:

```
typedef enum
{
  /** NONE is not used in actual traces */
  SARC_TRACE_NONE = 0,
  /** CRITICAL trace level */
  SARC_TRACE_CRITICAL = 1,
  /** ERROR trace level */
  SARC_TRACE_ERROR = 2,
  /** WARNING trace level */
  SARC_TRACE_WARNING = 3,
  /** INFO trace level */
  SARC_TRACE_INFO = 4,
  /** DEBUG trace level */
  SARC_TRACE_DEBUG = 5,
  /** TRACE trace level */
  SARC_TRACE_TRACE = 6
} SARC_TraceLevel;
```

11.3.2.2 Raise_Fatal_Error

This corresponds to the Flex_Raise_Fatal_Error generic function.

```
void #component_impl_name#_raise_fatal_error
   (#component_impl_name#_context* context,
    const char *format, ...);
```

Note: The error code cannot be given with this API. It is forced to the value 0.

The formatted string is truncated to 1024 characters, including a teminating null byte.

11.3.2.3 Display

This function allows the component to display arbitrary text on a text stream, which is not line-oriented, contrary to other logging functions. It can be considered as an abstract 'printf' function provided by the container. The text output may be stored, and/or directed to a local device, and/or sent remotely through a network.

```
void #component_impl_name#_display
   (#component_impl_name#_context* context,
    const char *format, ...);
```

The formatted string is truncated to 1024 characters, including a teminating null byte.

On some platforms, the implementation of this function may be empty. This has no incidence on the behaviour of the component.

11.4 Time Services

11.4.1 Get_Relative_Local_Time

```
SARC_int64 #component_impl_name#_get_local_time();
```

The returned time is expressed in nanoseconds.

11.4.2 Get_UTC_Time

Not applicable to this binding.

Note: UTC Time is an ECOA feature that is optional at component level. Component implementations that have the option 'needsSystemTime' set shall not use the present binding.

11.4.3 Get_Absolute_System_Time

```
SARC_int64 #component_impl_name#_get_time
   (#component_impl_name#_context* context);
```

The returned time is expressed in nanoseconds.

11.4.4 Get_Relative_Local_Time_Resolution

Not applicable to this binding.

Note: Access to time resolution is an ECOA feature that is optional at component level. Component implementations that have the option 'needsTimeResolution' set shall not use the present binding.

11.4.5 Get_UTC_Time_Resolution

Not applicable to this binding.

See note in previous section.

11.4.6 Get_Absolute_System_Time_Resolution

Not applicable to this binding.

See note in previous section.

11.5 Triggers

11.5.1 Trigger_Set

The C syntax for a component instance to set a trigger is:

```
SARC_Ecode #component_impl_name#_#trigger_name#_TRIGGER_set
   (#component_impl_name#_context* context,
    SARC_int64 delay);
```

The delay is expressed in nanoseconds.

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call, i.e., there is a pending trigger.

11.5.2 Trigger_Cancel

The C syntax for a component instance to cancel a trigger is:

```
SARC_Ecode #component_impl_name#_#trigger_name#_TRIGGER_cancel
   (#component_impl_name#_context* context);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call, i.e., there is no pending trigger.

11.6 Persistent Information management (PINFO)

11.6.1 PINFO read

The C syntax for a component instance to read persistent data (PINFO) is:

```
SARC_Ecode #component_impl_name#_#PINFOname#_ROM_read
   (#component_impl_name#_context* context,
    SARC_byte *memory_address,
    SARC_uint32 in_size,
    SARC_uint32 *out_size);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call.

11.6.2 PINFO write

Not applicable to this binding.

11.6.3 PINFO seek

The C syntax for a component instance to seek within persistent data (PINFO) is:

```
SARC_Ecode #component_impl_name#_#PINFOname#_ROM_seek
   (#component_impl_name#_context* context,
    SARC_int32 offset, SARC_RomOrigin whence,
    SARC_uint32 *new_position);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call.

11.7 Save Warm Start Context

Not applicable to this binding.

11.8 Supervisor components

This section is specific to [OPTION SUPERVISION].

When the component is of the SUPERVISOR kind, the C syntax for a component instance to command and control components, and variables is:

```
SARC_ExecutableStates #component_impl_name#_executable_status
   (#component_impl_name#_context* context,
    SARC_int32 executable_id);
```

```
void #component_impl_name#_executable_command
   (#component_impl_name#_context* context,
    SARC_int32 executable_id,
    SARC_ExecutablesCommands command);

void #component_impl_name#_component_state_command
   (#component_impl_name#_context* context,
    SARC_int32 p_instance,
    SARC_LifeCycleShift p_shift);

SARC_LifeCycleState #component_impl_name#_component_status
   (#component_impl_name#_context* context,
    SARC_int32 p_instance);

SARC_Ecode #component_impl_name#_get_variable_#variable_name#
  (#component_impl_name#_context *context,
   #variable_type# *value);

SARC_Ecode #component_impl_name#_set_variable_#variable_name#
  (#component_impl_name#_context *context,
   #variable_type# value);
```

12 Container Types

This section contains details of the data types that comprise the container API i.e. the data types that can be used by a component.

12.1.1 Versioned Data Handles

This section contains the C syntax in order to define data handles for versioned data operations defined in the Container Interface.

```
/*
 * The following is the data handle structure associated to the data operation
 * called #operation_name# of data-type #type_name#
 */
typedef struct {
   /* pointer to the local copy of the data */
   #type_name# * ptr;
   /* version counter modified by the infrastructure
      each time the data is updated */
   SARC_int32 ref;
} t_#operation_name#_handle;
```

13 Default Values

Not applicable to the C Binding.

14 External Interface

Not applicable to this Binding, since it does not implement [OPTION EXTERNAL INTEFACE].

15 PeriodicTriggerManager Components

There is no specific API for PeriodicTriggerManager components, since these components are entirely managed by the infrastructure.

16 External Components

This section contains the C syntax for the "External" special component kind.

The C syntax for the external routine (code executed by the external thread) of an External Component is:

```
void #component_impl_name#_external_routine
   (#component_impl_name#_context* context);
```

This definition is in **#component_implementation#**.h.

The C syntax for the functions to allow starting and stopping the external thread is:

```
SARC_Ecode #component_impl_name#_start_external_thread
(#component_impl_name#_context* context);
SARC_Ecode #component_impl_name#_stop_external_thread
(#component_impl_name#_context* context);
```

SARC_Ecode return value is one among the following set:

- SARC_OK: operation call performed properly;
- SARC_KO: failed operation call.

This definition is in **#component_implementation#**_container.h.

17 Reference C Header

The following file contains definitions that are included from any header file defined in §6.2.

It is given as an example only. In case of inconsistency between the above text and the following file, the above text has precedence.

```
#ifndef SOFTARC_H
#define SOFTARC_H

#ifdef __cplusplus
extern "C"
```

```
{
#endif

/**
 * Boolean type - 1 byte long
 */
typedef unsigned char SARC_boolean8;

/** Boolean value for logical false */
#define SARC_FALSE 0U
/** Boolean value for logical true */
#define SARC_TRUE 1U

typedef char SARC_Byte;

/** Scalar type for characters - 1 byte long */
typedef char SARC_char8;

/** Scalar type for 1-byte long signed integers */
typedef signed char SARC_int8;

/** Scalar type for 1-byte long unsigned integers */
typedef unsigned char SARC_uint8;

/** Scalar type for 2-byte long signed integers */
typedef signed short int SARC_int16;

/** Scalar type for 2-byte long unsigned integers */
typedef unsigned short int SARC_uint16;

/** Scalar type for 4-byte long signed integers */
typedef signed int SARC_int32;

/** Scalar type for 4-byte long unsigned integers */
typedef unsigned int SARC_uint32;

/** Scalar type for 8-byte long signed integers */
typedef signed long long SARC_int64;

/** Scalar type for 8-byte long unsigned integers */
typedef unsigned long long SARC_uint64;

/** Scalar type for IEEE-754 simple precision floating point numbers */
typedef float SARC_float32;
```

```
/** Scalar type for IEEE-754 double precision floating point numbers */
typedef double SARC_double64;


typedef enum
{
  /** Used when function behaved as expected */
  SARC_SUCCESS = 0,
  /** Most of the time, abnormal behaviour results in a FAILURE */
  SARC_FAILURE = 1,
  /** When a function returns because its execution time slot has expired */
  SARC_TIMEOUT = 7,
  /** Alias for SARC_SUCCESS - deprecated */
  SARC_OK = SARC_SUCCESS,
  /** Alias for SARC_FAILURE - deprecated */
  SARC_KO = SARC_FAILURE,
  /** Used by SOFTARC container functions to link FAILURE to its inputs */
  SARC_INVALID_IN_PARAMETER = 2,
  /** Used by SOFTARC container functions to link FAILURE to its outputs */
  SARC_INVALID_OUT_PARAMETER = 3,
  /**
   * Used by SOFTARC container functions regarding data management to link
   * FAILURE to its inputs
   */
  SARC_INVALID_DATA = 4
} SARC_Ecode;


/**
 * This enumerated type describes the possible ways a data
 * handle can be initialized.
 */
typedef enum
{
    /**
     * The memory area pointed by the handle has no specific value
     * Use this by default in order to have better performances.
     */
  SARC_DATA_NO_VALUE,

    /**
     * The memory area pointed by the handle is filled with the
     * current data value. This mode has lower performances, but
     * allows access the current data version before modifying it.
     */
  SARC_DATA_CURRENT_VERSION
} SARC_DataValue;


/** Reference position used when moving the reading head */
typedef enum
{
 /** Beginning of ROM */
  SARC_ROM_ORIGIN_START,
  /** Current position */
  SARC_ROM_ORIGIN_CURRENT,
  /** End of ROM */
  SARC_ROM_ORIGIN_END
} SARC_RomOrigin;


typedef SARC_uint32 SARC_LifeCycleState;

/** UNAVAILABLE Instance state */
#define SARC_LIFE_CYCLE_STATE_UNAVAILABLE 0x00
/** IDLE Instance state */
#define SARC_LIFE_CYCLE_STATE_IDLE 0x01
/** READY Instance state */
#define SARC_LIFE_CYCLE_STATE_READY 0x02
/** RUNNING Instance state */
#define SARC_LIFE_CYCLE_STATE_RUNNING 0x03


typedef SARC_uint32 SARC_LifeCycleShift;

/** INITIALIZE transition */
#define SARC_LIFE_CYCLE_SHIFT_INITIALIZE 0x02
/** START transition */
#define SARC_LIFE_CYCLE_SHIFT_START 0x03
/** RESET transition */
#define SARC_LIFE_CYCLE_SHIFT_RESET 0x04
/** STOP transition */
#define SARC_LIFE_CYCLE_SHIFT_STOP 0x05
/** SHUTDOWN transition */
#define SARC_LIFE_CYCLE_SHIFT_SHUTDOWN 0x06
/** KILL transition */
#define SARC_LIFE_CYCLE_SHIFT_KILL 0x07
```

```
typedef SARC_uint32 SARC_ExecutableStates;

/** Tag for unavailable executable status */
#define SARC_STATUS_EXECUTABLE_NULL 0
/** Tag for running executable status */
#define SARC_STATUS_EXECUTABLE_LAUNCHED 1


typedef SARC_uint32 SARC_ExecutablesCommands;

#define SARC_PANEL_COMMAND_LAUNCH 1
#define SARC_PANEL_COMMAND_KILL 2


#ifdef __cplusplus
}
#endif

#endif                              /* SOFTARC_H */
```

18 Compatibility with ECOA Options

18.1 Compatibility with Options defined in the ECOA Standard

The following table indicates, for each optional functionnality defined in the ECOA Standard (taken from [Part 5] document), whether it is supported or not by this binding.

- YES: the option is supported by this binding
- NO: the option is not supported by this binding
- N/A: Not Applicable. The option has no impact on bindings.

Name of Option	**Supported by this binding**
[OPTION SUPERVISOR COMPONENTS]	YES
[OPTION ELI]	N/A
[OPTION FAULT HANDLING]	NO
[OPTION MULTI APP ASSEMBLY]	N/A
[OPTION DYNAMIC TRIGGER MANAGER]	N/A
[OPTION UINT64]	YES
[OPTION INT64]	YES
[OPTION PINFO WRITE]	NO

[OPTION WARM START CONTEXT]	YES
[OPTION AUTO START EXTERNAL TASK]	N/A
[OPTION SYSTEM TIME]	YES
[OPTION UTC TIME]	NO

18.2 Compatibility with Component Implementation Options

The following table gives the possible values of each option listed in [AS Part 7], §6.1.3 Component Implementation, for components that use the SOFTARC C Binding:

Component Implementation Option	**Allowed values for component implementations using this Binding**
needsLocalTime	false, true
needsSystemTime	false, true
needsUTCTime	false
needsTimeResolution	false
hasReset	true
isFaultHandler	false
autostartExternalThread	false, true
hasWarmStartContext	false, true