Component Architecture Debug Interface v2.0

Developer Guide

ARM®

Component Architecture Debug Interface v2.0
Developer Guide

Copyright © 2008-2012 ARM. All rights reserved.

Release Information

Contents
Component Architecture Debug Interface v2.0 Developer Guide

Preface

This preface introduces the *Component Architecture Debug Interface v2.0 Developer Guide*. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page viii.

About this book

This document describes the class hierarchy and programming interfaces for version 2.0 of the *Component Architecture Debug Interface* (CADI). It is intended for users writing applications and debug tools that use the CADI interface.

Intended audience

This document has been written for experienced hardware and software developers to design systems or components.

Users must be familiar with the basic concepts of C++ such as classes and inheritance.

Using this book

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this for an introduction to designing components.

Chapter 2 *The Target Connection Mechanism*

Read this for a description of how to implement the target connection mechanism that connects to a simulation.

Chapter 3 *Using the CADI Interface Methods from a Debugger*

Read this for a description of how to use an application, typically a debugger, to access and control the target by calling methods supplied by the CADI interface.

Chapter 4 *The CADI Extension Mechanism*

Read this for a description of the mechanism available to customize and extend the CADI interface.

Appendix A *Class Reference*

Read this for a description of all of the classes used in the CADI interface and gives a brief description of their methods.

Appendix B *Data Structures Used by the CADI Interface*

Read this for a description of the main data structures within the CADI classes.

Glossary

The *ARM Glossary* is a list of terms used in ARM documentation, together with definitions for those terms. The *ARM Glossary* does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See *ARM Glossary*, http://infocenter.arm.com/help/topic/com.arm.doc.aeg0014-/index.html.

Typographical conventions

The typographical conventions are:

italic   Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

bold   Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

monospace	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
<u>mono</u>space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
monospace italic	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
monospace bold	Denotes language keywords when used outside example code.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Additional reading

This section lists related publications by ARM and by third parties.

See Infocenter, http://infocenter.arm.com for access to ARM documentation.

ARM publications

This book contains information that is specific to this product. The following publications provide reference information about the ARM architecture:

*   *AMBA™ Specification* (ARM IHI 0011)
*   *ARM Architecture Reference Manuals*,
    http://infocenter.arm.com/help/topic/com.arm.doc.set.architecture/index.html.

The following publications provide information about related ARM products:

*   *Model Debugger for Fast Models User Guide* (ARM DUI 0314).

Feedback

ARM welcomes feedback on this product and its documentation.

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

• The product name.

• The product revision or version.

• An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to errata@arm.com. Give:
• the title
• the book number, ARM DUI 0444J
• the page numbers to which your comments apply
• a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the *Component Architecture Debug Interface* (CADI). It contains the following sections:

- *About the Component Architecture Debug Interface* on page 1-2
- *The class hierarchy* on page 1-4
- *CADI classes used to connect to a simulation* on page 1-7.

1.1 About the Component Architecture Debug Interface

The *Component Architecture Debug Interface* is a C++ API that enables convenient and accurate debugging of complex System-on-Chip (SoC) simulation platforms. It enables a caller, typically a debugger, to:

- connect to an existing simulation or instantiate a new simulation
- attach to one of the simulation targets
- control the execution of a connected target
- observe and manipulate simulated hardware resources
- display the contents of registers and memory in the simulation targets
- obtain valuable disassembly or profiling information.

——— **Note** ———

CADI can be used with simulation targets at any level of abstraction such as, for example, instruction-accurate simulation or cycle-accurate simulation platforms.

Because CADI is widely used within ARM solutions, developers can:

- integrate ARM models into their own design methodologies.

- benefit from analyzing their simulation platforms with ARM development tools such as *Development Studio* (DS-5™) or Model Debugger for Fast Models.

CADI enables interaction with a third-party debugger to:

- integrate a processor model with established user base for an existing debugger
- support an architecture that has only a limited range of native debuggers.

A CADI implementation provides many technical benefits such as:

Retargetability

> The interface exposes sufficient information on a target to enable describing its behavior and hardware resources. The caller does not require additional description files to analyze or visualize hardware components.

Semihosting

> Semihosting calls establish a channel for input to and output from an application running on the target. This enables callers to:
> - interact with the target
> - redirect target input and output to the host system the simulation platform is running on.

Callbacks Callback methods enable the use of asynchronous method calls to the target that minimize the impact on the behavior of the target. The target is blocked by a single caller for only a short period.

> Synchronous calls through the interface lock out other callers until the call has ended. This is often undesirable behavior, If, for example, one debugger is executing a command on the target, another debugger is blocked from stopping the target.

Compiler support

> The design of CADI v2.0 classes and data types avoids method calls that pass the ownership of heap memory objects between the caller and the target. This enables interaction between tools and models compiled with different compilers.

Optional implementation

Functionally separated parts of CADI can be optionally implemented. This applies to both single method calls of the common CADI interface and to those in independent classes of the CADI class hierarchy.

1.2 The class hierarchy

Figure 1-1 shows the CADI class hierarchy.

――――― **Note** ―――――

In this guide, references to the Component Architecture Debug Interface (CADI) in total are distinguished from references to the individual `CADI` class by using a monospace font for the `CADI` class.

A CADI simulation is the simulation of a platform that can be accessed by using an implementation of the CADI interface. Typically at least one platform component exposes an implementation of class `CADI`. This component can be referred to as a CADI target.

The methods in the top-level classes are pure virtual. The methods of the lowest-level user classes implement the component-specific behavior.
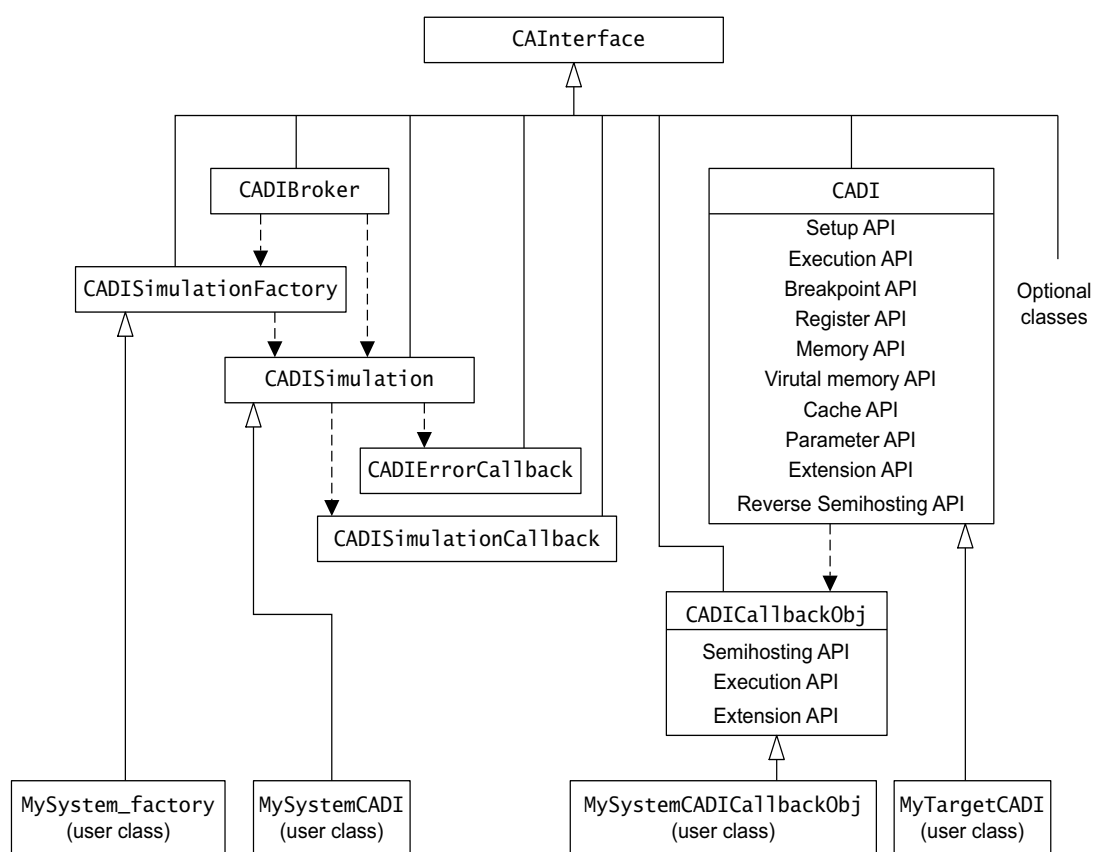
Figure 1-1 CADI class overview

The majority of the CADI functionality is exposed through the following classes:

CADI   The `CADI` object handles the requests from the outside world into the target.

                `CADI` objects are implemented by the models.

                A pointer to the `CADI` object can be obtained from the `GetTarget()` method of the `CADISimulation` class.

CADICallbackObj

                The `CADICallbackObj` handles the calls made by the target towards the debugger to, for example, indicate state changes in the model.

CADICallbackObj objects must be implemented by the debugger and registered with the target.

The CADICallbackObj is also used for semihosting requests. Instead of requiring the simulation of a full operating system, CADI provides the option to forward the console operations from the target to the host operating system.

Most of the functionality available through CADICallbackObj could be obtained by polling the state of the target model each cycle through the regular CADI interface. It is more efficient however to have the target use the callbacks as required. Using callbacks eliminates the large overhead that results from frequent polling calls.

The model can call the callback methods at any time during simulation. ARM recommends, however, that the callback handlers do as little processing as possible and, for example, only set flags for later processing. The remaining processing can be done by the debugger without delaying the simulation.

There are several conceptually distinct parts of the CADI interface:

CAInterface class

This class is the base class for all CADI classes and enables creation and use of software models built around components and interfaces. See *The CAInterface class* on page A-2.

Simulation and factory classes

These classes described in the following sections provide the mechanism for creating and running simulations:
- *The CADIBroker class* on page A-4
- *The CADISimulationFactory class* on page A-8
- *The CADIErrorCallback class* on page A-11
- *The CADISimulationCallback class* on page A-12
- *The CADISimulation class* on page A-13.

CADI class The methods in this class (described in *The CADI class* on page A-20) provide the main interfaces for configuring and running the target. Use these methods to:
- set up the target
- control target execution
- set breakpoints
- extent the standard interface
- access registers
- access memory
- access cache.

CADICallbackObj class

The methods in this callback class (described in *The CADICallbackObj class* on page A-16) enable the target to communicate with the debugger and:
- provide semihosting I/O
- notify the debugger of a change in execution mode in the target
- support extensions to the standard interface.

CADI disassembler classes

If the component supports disassembly, the disassembly interface can obtain the disassembly during a simulation. These classes are described in the following sections:
- *The CADIDisassemblerCB class* on page A-46

- *The CADIDisassembler class* on page A-48).

CADI profiling classes

The profiler class enables you to record and monitor profile information related to the debugging session. These classes are described in the following sections:

- *The CADIProfilingCallbacks class* on page A-52
- *The CADIProfiling class* on page A-53).

——— **Note** ———

The CADI Profiling classes are not supported by the Fast Model Portfolio processor models. This guide, therefore, contains only a high-level overview of the profiling classes.

——— **Note** ———

For more detail on individual classes, see Appendix A *Class Reference*.

See Appendix B *Data Structures Used by the CADI Interface* and the `CADITypes.h` file for definitions of enumerations and data structures that are used with the CADI interface.

1.3 CADI classes used to connect to a simulation

Figure 1-2 shows the relation of CADI interface classes in the target connection mechanism.
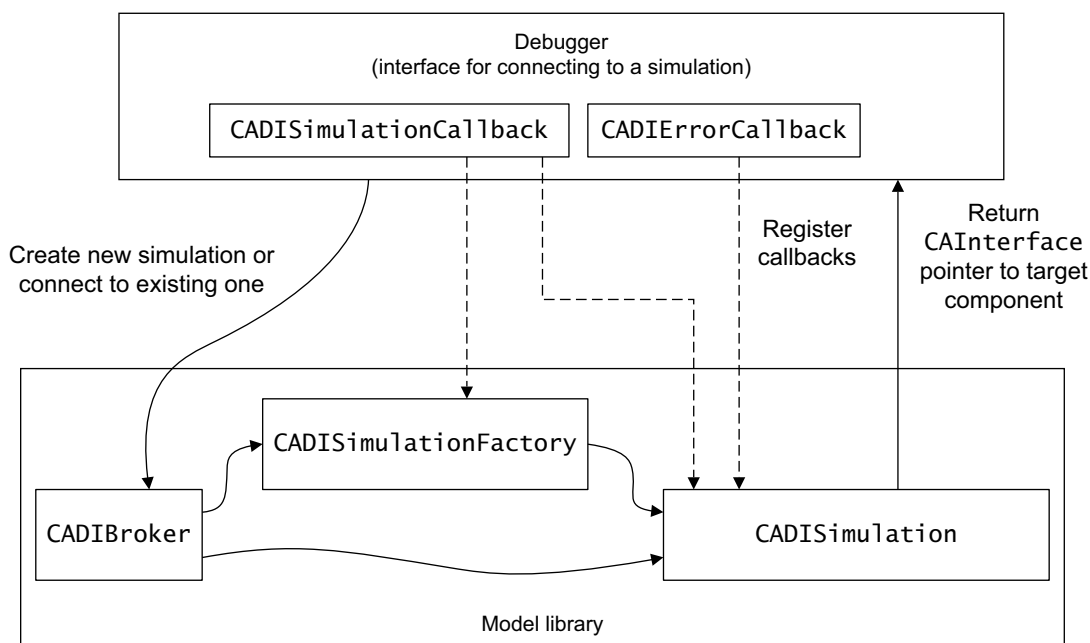
Figure 1-2 Relationship between CADI interface classes used to connect to a target

Each interface class is derived from `CAInterface` to enable compatibility checks and the extension mechanism. See *The CAInterface class* on page A-2.

The `CADIBroker` class manages the connection to a CADI simulation and consequently to a target. It provides a CADI simulation by either:

- returning an existing simulation that can be connected to. A `CADISimulation` object is directly returned.

- obtaining a CADI simulation factory that instantiates a CADI simulation. A pointer a `CADISimulationFactory` object is selected and obtained. If a CADI factory creates a simulation, it transfers the pointer to the new simulation to the broker

The `CADISimulation` class interacts with the `CADISimulationCallback` and `CADDIErrorCallback` callback classes. An object of each of these classes must be registered to it. Pointers to the callback objects are forwarded to the simulation and used for asynchronous communication between the target and debugger.

It is necessary to unregister the callback before ending the simulation. This avoids problems that might result from disconnecting from a simulation without shutting it down.

1.3.1 CADI classes used to control the simulation target

The `CADISimulation` method `GetTarget()` returns a pointer, of type `CAInterface`, to the required target component. After calling its `ObtainInterface()` method to validate interface compatibility, the target can convert the pointer to the wanted interface type.

The targeted interface acquisition is depicted in Figure 1-3 on page 1-8. All of the objects shown derive from `CAInterface`.

The standard CADI interfaces that can be obtained from the target pointer are CADI, CADIDisassembler, CADIProfiling, or a type that corresponds to a custom extension. The type is typically CADI or CADIDisassembler. These interfaces might not, however, be implemented for a target (see *Optional implementation*).

Chapter 4 *The CADI Extension Mechanism* describes adding interface extensions. Extensions are in addition to the standard types. Dedicated callback objects must be registered. Communication is typically asynchronous into both directions, but the caller must manage synchronization of calls and any associated callbacks.
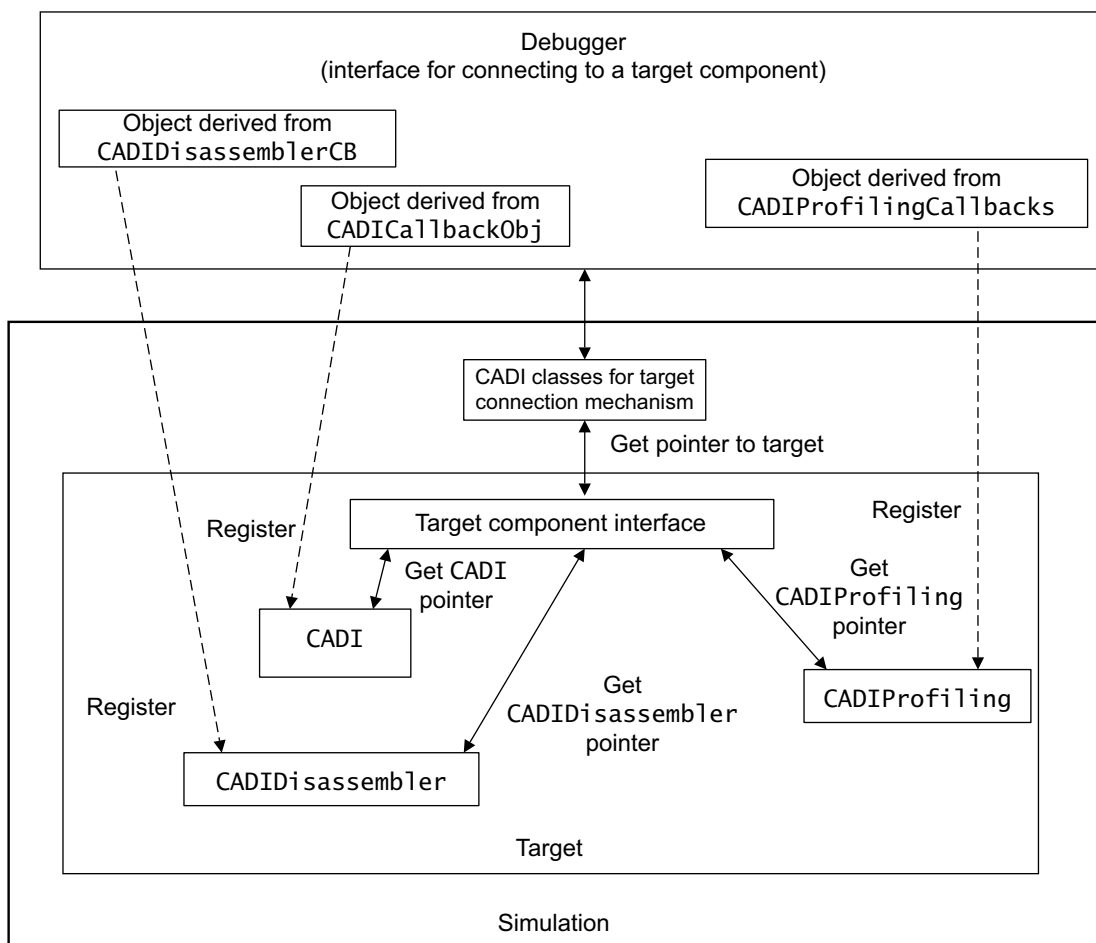
Figure 1-3 Relationship between CADI classes for target interface

1.3.2   Optional implementation

A given CADI target might only implement a subset of the CADI interface methods. For API implementation details for the CADI targets of a specific model, see the model documentation.

A target for a memory model, for example, only requires the Memory API and does not require the Register API or the Disassembly API.

The disassembler and profiler classes shown in Figure 1-4 on page 1-9 are optional.

Standard classes that might not be implemented
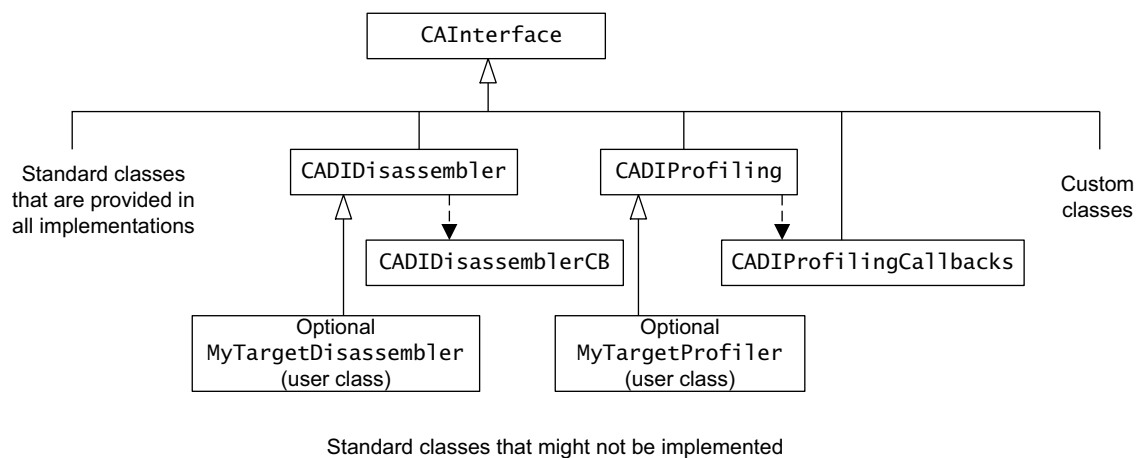
Figure 1-4 Optional CADI classes

The Breakpoint and Execution APIs might not be implemented by all processor models. Unimplemented methods that never return successfully return `CADI_STATUS_CmdNotSupported`.

Chapter 2
The Target Connection Mechanism

This chapter describes the target connection mechanism. It contains the following sections:

2.1   Introduction to the target connection mechanism

CADI 2.0 provides two well-defined mechanisms for creating a connection to a target:

- connecting to an existing simulation that was, for example, started from another tool
- instantiating a simulation and connecting to one or more of its components.

The interface also provides a clean way to disconnect from a target.

The connection mechanism consists of a set of interface classes that must be implemented. Figure 2-1 and Figure 2-2 on page 2-3 show the steps required to establish a connection.

Advantages of the CADI 2.0 connection mechanism over previous CADI versions are the ability to:

- create multiple instances of the same `CADISimulation`

- fully configure and instantiate a simulation platform before connecting to one of its components

- obtain an extension interface.
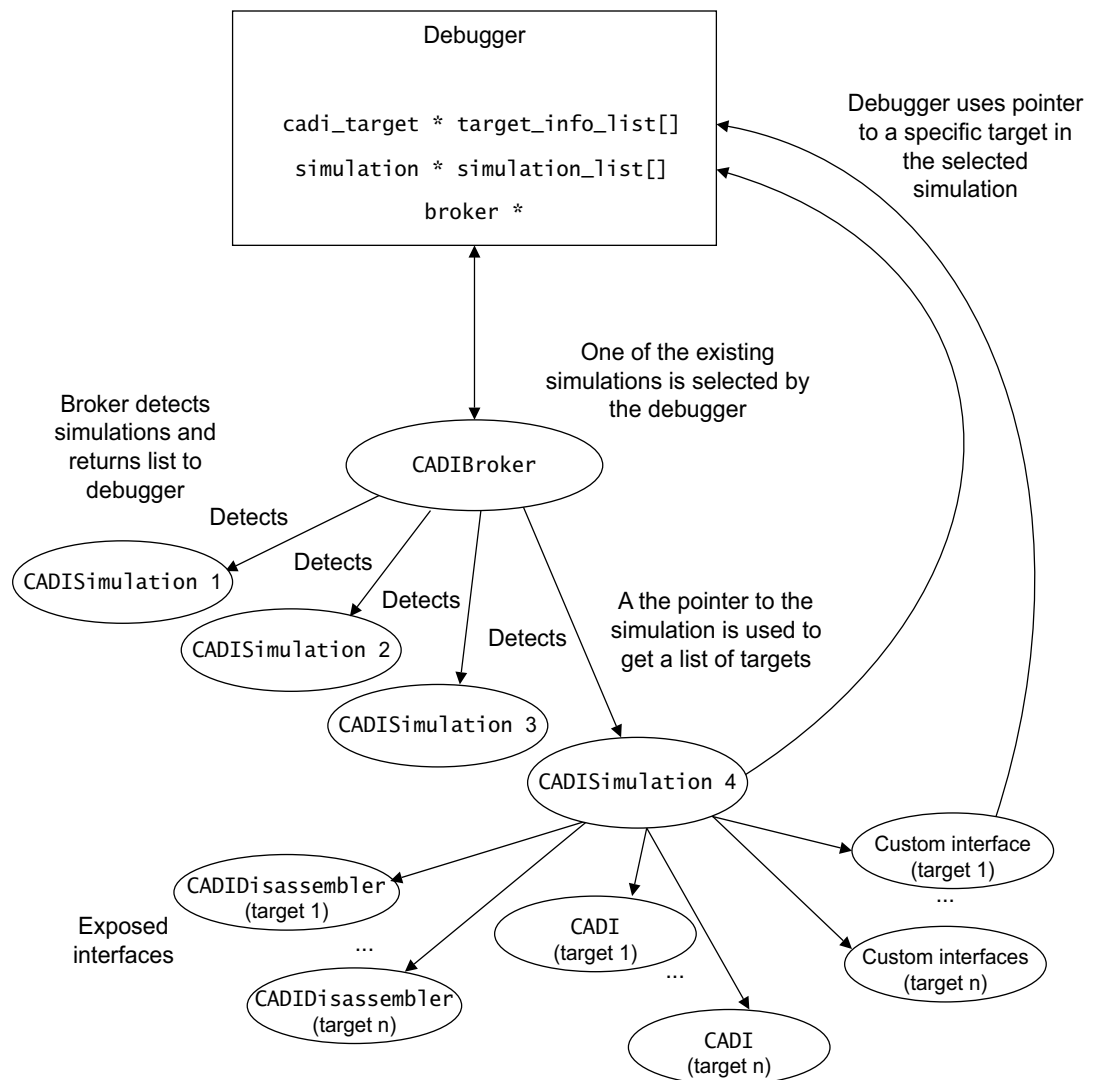
Figure 2-1 Connection sequence for existing simulation
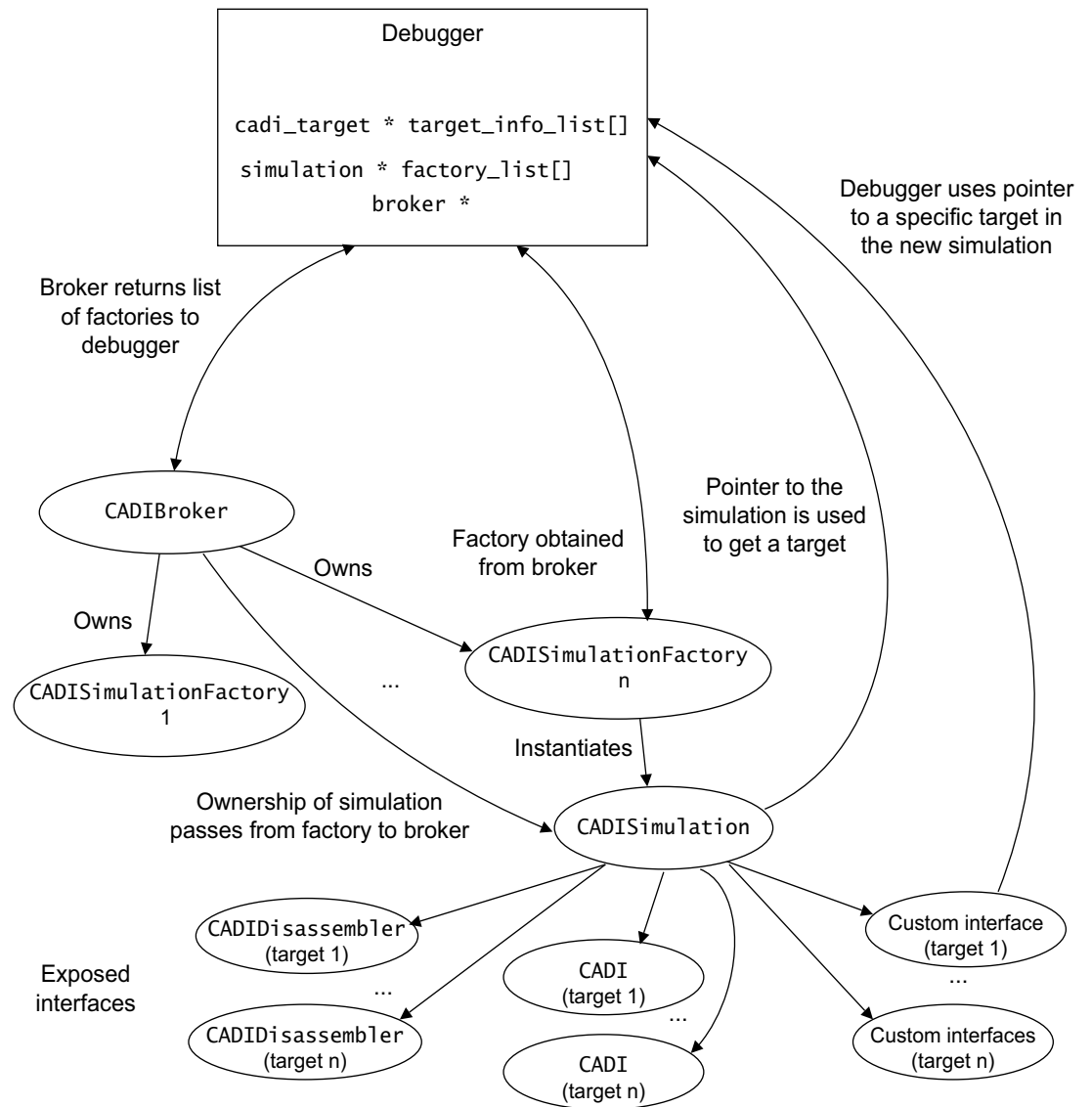
Figure 2-2 Connection sequence for new simulation

2.2 Implementing the target connection mechanism

Implementing the target side of the CADI target connection mechanism requires implementing:

- one global function
- the corresponding interface classes and their methods.

There are specific requirements for the implementation of each class. These are described in the following sections:

- *CADIBroker*
- *CADISimulationFactory* on page 2-6
- *CADISimulation* on page 2-7
- *ObtainInterface()* on page 2-7
- *Callback objects* on page 2-8.

2.2.1 CADIBroker

The CADIBroker is the central element of the target connection mechanism. It establishes the connection to existing simulations and the instantiation of new simulations.

The CreateCADIBroker() function in a model library indicates the presence of a CADI interface. The function returns a pointer to a CADIBroker object. The function can be implemented in one of two ways depending on how the broker is implemented in the addressed library:

- the CADI broker is a singleton object and the call returns a pointer to it

- a new CADI broker object is instantiated and the call returns a pointer to it as shown in Example 2-1.

Example 2-1 Obtaining a pointer from a new CADIBroker object

```
CADI_WEXP eslapi::CADIBroker* CreateCADIBroker()
{
    return (new MyCADIBroker());
}
```

The following mechanisms support connecting to a simulation:

Connecting to an existing simulation

The broker returns details of all running simulations. This information is used to create a connection to an existing simulation.

Create a new simulation and connect to it

The broker returns a list of simulation factories. This information is used to instantiate a new simulation.

For both connection methods, the debugger must cleanly disconnect from running simulations. Disconnection is required for:

- shutting down a simulation because of an event in the simulation or debugger

- ending the debugger session, but keeping alive the simulation for other current or future debug connections.

Selecting and connecting to a simulation

The `SelectSimulation()` method receives two pointers to callback objects (`CADISimulationCallback` and `CADIErrorCallback`) and an array containing the enable vector for `CADISimulationCallback`. These callback objects might be used during the `SelectSimulation()` call if, for example, the simulation wants to shut down at exactly at the same moment the debugger starts connecting to it.

`SelectSimulation()` also forwards the callback objects to the returned CADI simulation.

The `CADISimulationCallback` object provides the `CADISimulation` object with a mechanism to guarantee a clean disconnect of the debugger.

This way of connecting is typically associated with a server-client-technique where the client is represented by a CADI broker. The server might be, for example, directly embedded into a simulation platform or implemented within an environment that runs the simulation.

Retrieving a list of simulation factories and instantiating a new simulation

The broker returns a list of pointers to the available simulation factories. The broker controls the simulation factory objects. It must destroy them before it is released.

After a CADI simulation factory is obtained, it is used to establish a connection to a newly instantiated CADI simulation:

1.   The simulation is instantiated.
2.   The new simulation returns a pointer to the corresponding `CADISimulation` object.
3.   The pointer is used to select a target in the simulation and connect to it.

In addition to managing the simulation factories, the CADI broker is also responsible for the CADI simulation objects (especially if the simulation objects are directly owned by the broker).

For more information on this connection mechanism, see the *CADISimulationFactory* on page 2-6.

Setting preprocessor defines

In Example 2-1 on page 2-4, the `MyCADIBroker` class implements the CADI broker. The macro `CADI_WEXP` preceding the function prototype is only relevant for exporting this symbol from a Windows DLL:

*   Setting the preprocessor define `EXPORT_CADI` sets `CADI_WEXP` to `__declspec(dllexport)` that makes the `CreateCADIBroker()` function call to be an exported symbol for the built model DLL.

*   Not setting the preprocessor defines `EXPORT_CADI` and `NO_IMPORT_CADI` causes `CADI_WEXP` to be set to `__declspec(dllimport)`. This makes the `CreateCADIBroker()` function call an imported symbol for the built model DLL.

*   Not setting the preprocessor define `EXPORT_CADI`, but setting the preprocessor define `NO_IMPORT_CADI`, defines `CADI_WEXP` to be empty.

For more information on these settings, see the `CADICommon.h` file.

A similar scheme applies to the macro `ESLAPI_WEXP` and the preprocessor defines `EXPORT_ESLAPI` and `NO_IMPORT_ESLAPI`. This macro declares the symbol attributes for `CAInterface`. Because `CADI` is derived from `CAInterface`, these preprocessor defines must be set if building a model DLL that exposes a CADI interface.

2.2.2 CADISimulationFactory

The `CADISimulationFactory` creates a new CADI simulation. The simulation factory:

- provides basic information (name and a brief description) about the simulation associated to it

- exposes information on the available instantiation time parameters.

> ——— **Note** ———
> During the process of creating a CADI simulation, you can either configure:
> — all of the instantiation-time parameters for the entire platform
> — only the components in the simulation.

A typical platform is hierarchical in design and contains multiple components. The name of a parameter indicates its ownership of a dedicated sub-component. A dedicated specifier represents each hierarchical level and its corresponding component:

- the identifier is typically the name of the component at that level
- the levels are separated by dots (.)
- the last element of a specification string is the parameter name itself.

For example, the `size` parameter for a memory component named `mem` in the processor component of a system named `socsystem` is `socsystem.core.mem.size`.

> ——— **Note** ———
> In this guide, the term *target* is typically used instead of *component*. Both terms describe a sub-system, or a single component, in a platform.

During instantiation of a CADI simulation, the corresponding interface method receives the parameters:

- it is not mandatory to set all parameters

- if the caller does not provide a value for a parameter, the simulation factory uses the default value retrieved from the parameter information

- parameters forwarded to the simulation during instantiation are not required to be in the same order as the received parameter list

- the forwarded parameter list is not required to be complete

- the caller must signal the end of a list by adding an additional terminating item with the parameter ID `0xFFFFFFFF`.

> ——— **Note** ———
> The terminating ID of `0xFFFFFFFF` is the same as `static_cast<uint32_t>(-1)`.

As with `SelectSimulation()`, the `Instantiate()` method can receive pointers to `CADISimulationCallback` and `CADIErrorCallback` objects. The pointers are registered to the CADI simulation returned to the caller. These callbacks are used, for example, to send messages from the factory to the caller during instantiation.

> ——— **Caution** ———
> A CADI simulation factory is intended to exist only temporarily. As soon as the required CADI simulation is created, the `Release()` method must be called to release the factory.

Because of the temporary existence of the factory, `CADIBroker` becomes the owner of the simulation.

2.2.3 CADISimulation

The `CADISimulation` class represents the connection to a simulated platform and provides information about platform targets that expose a CADI interface. Querying this object returns a list with an element for each target. The descriptions include:

- the target ID that must be used for interaction between the caller and CADI simulation related to this target

- fundamental properties that might have a major impact on the behavior of an attached debugger (for example if the target is able to execute software).

The caller uses the returned information to select a target. To retrieve a pointer to the corresponding target, call the `GetTarget()` method of the CADISimulation. The returned pointer is to `CAInterface` in the base class of the CADI interface.

As with other classes in the target connection mechanism, `CADISimulation` contains a `Release()` method to disconnect the caller from a simulation. After `Release()` is called, an attached debugger must not address the simulation or a target previously obtained from the simulation. Calling a released simulation might raise an access violation because the connected target or simulation, and the associated `CADI` object, might already be destroyed. The CADI simulation object owns all target interfaces associated with the simulation and is therefore responsible for their creation and destruction.

A major difference between the `Release()` call of `CADISimulation` and those of the other classes is the `shutdown` parameter:

- If `true`, the implementation for this method must manage shutting down the connected simulation. Shutdown includes informing other connected callers about the shutdown and waiting for them to acknowledge the request by calling `Release()` themselves.

- If `false`, a simulation might be kept alive after disconnection. This might be the case if the debugger is one of multiple callers and there is no requirement to enforce a shutdown on disconnect.

Typically, a `CADISimulationCallback` object and a `CADIErrorCallback` object are registered to a `CADISimulation` through the corresponding methods that established the connection. Dedicated methods are provided to add additional callback objects to the simulation.

2.2.4 ObtainInterface()

`ObtainInterface()` must be implemented for all of the CADI classes used in the target connection mechanism. `ObtainInterface()` identifies the availability of a specific interface and the version of the interface. It performs a compatibility check for the caller:

- The implementation first compares the interface name and revision number with those forwarded through the method call.

   If no compatible interface is found, the same checks are performed for base classes if they are available.

- If the checks are successful and the requested interface is available, a `CAInterface` pointer is returned. The pointer type must be converted to the interface class that was actually requested.

- If no compatible interface is found, a `NULL` pointer is returned.

See *Implementation example for extending the target-side* on page 4-3 for `ObtainInterface()` implementation examples.

2.2.5 Callback objects

The target connection mechanism of CADI uses the following callback classes:

`CADIErrorCallback`

> `CADIErrorCallback` is primarily used to report errors from a simulation to the registered caller. This manages errors occurring during the target connection phase and during the simulation itself.

`CADISimulationCallback`

> `CADISimulationCallback` is required for system-wide communication from a CADI simulation to the caller.
>
> Callback methods of this class are of special importance for the CADI interface because they are required to guarantee a clean disconnection of a caller from a target or simulation and, if required, a clean shutdown of the simulation. A shutdown requires the `simShutdown()` and `simKilled()` methods:
>
> - `simShutdown()` indicates to a caller that the simulation is shutting down. That might be the result of a simulation being requested to shut down by an internal event or by another attached debugger receiving this callback.
>
>   A caller must unregister all callback objects and release all obtained interface pointers acquired during target connection.
>
> - If it is not possible to cleanly disconnect and shut down the simulation, the `simKilled()` callback must be called. This tells the caller that the interface no longer exists because of, for example, a hardware failure or memory access error.
>
>   After `simKilled()` is received, a caller must not access the corresponding simulation pointer or objects owned by the simulation.
>
> See *Disconnecting from a target* on page 2-21 for examples of disconnection and shutdown scenarios.

2.3 Connecting to a simulation

This section describes in detail how to connect to a CADI target and how to use the required factory mechanism.

Unless otherwise specified, the instructions apply to either:
- connecting to an existing simulation
- instantiating a new simulation.

2.3.1 Opening the model library

The first step to establish a connection to a CADI simulation is opening the dynamic library that implements the CADI interface. This is not necessarily the same library that implements the simulation itself.

If remotely connecting to a simulation, the opened dynamic library must translate the calls arriving at the CADI interface into a format that can be transferred through a common interface such as, for example, TCP.

2.3.2 Creating the CADIBroker

After opening the library, the next step in establishing a target connection is calling `CreateCADIBroker()` as shown in Figure 2-3. This call instantiates a new broker and returns a pointer to it.

If the library implements the broker as a singleton object, `CreateCADIBroker()` returns a pointer to the singleton object.
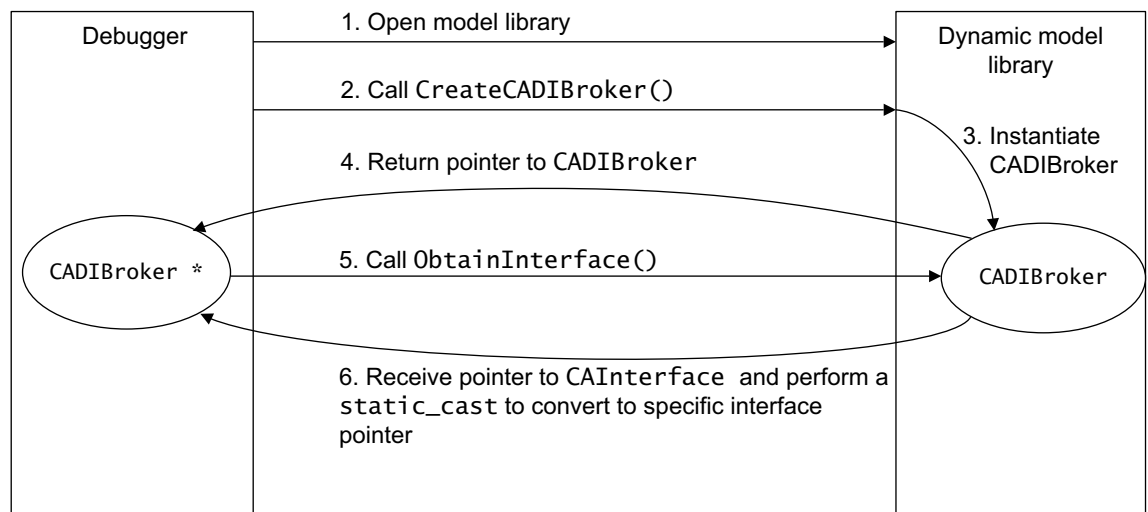
Figure 2-3 Creating a CADIBroker

The `ObtainInterface()` method from the returned broker must be called to verify compatibility with the caller. The obtained `CAInterface` pointer must be converted back to a `CADIBroker` pointer using a `static_cast` as shown in Example 2-2:

Example 2-2 Creating a CADIBroker

```
//get "CreateCADIBroker" symbol from dynamic library "dll"
void* entry = lookup_symbol(dll, "CreateCADIBroker");
CADIBroker* cadi_broker =
```

```
                    ((*eslapi::CreateCADIBroker_t)entry)();
//compatibility check
CAInterface* ca_interface;
if_name_t ifName = "eslapi.CADIBroker2";
if_rev_t minRev = 0;
if_rev_t actualRev = 0;
ca_interface = cadi_broker->ObtainInterface(ifName,
                                            minRev,
                                            &actualRev);

if (ca_interface == NULL)
{
   //something went wrong, handle it...
}
else
{
   cadi_broker = static_cast<CADIBroker*>(ca_interface);
}
     ...
```

For more information on the alternatives for connecting to a simulation, see:

2.4 Using GetSimulationFactories()

One way to establish a connection to a simulation target within CADI is to instantiate a CADI simulation and to connect to one of its targets.

To retrieve the list of available CADI simulation factories, the caller must execute the GetSimulationFactories() method in the CADI broker as shown in Figure 2-4. The result of this call is an array of CADISimulationFactory pointers.

The list of simulation factories is static for a CADI broker, therefore it is only required to retrieve the list once at the beginning of a debug session.

────── **Note** ──────

The caller is responsible for releasing, but not deleting, all obtained simulation factory objects. It is not sufficient to release only those that have been used to instantiate a simulation.
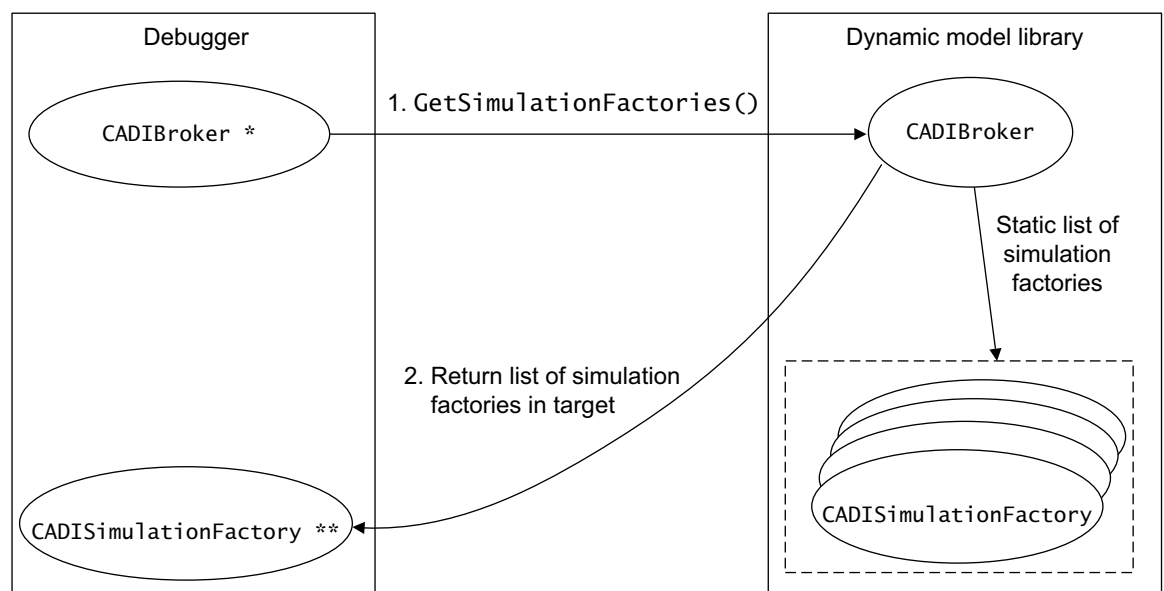
Figure 2-4 Getting the CADI simulation factories

After retrieving the list of available simulation factories, the next step is to call the ObtainInterface() method of the CADI broker to check the compatibility of the requested factory. A static_cast() is required for the interface as described in *Creating the CADIBroker on page 2-9*.

After obtaining the appropriate CADI simulation factory, the caller must prepare the configuration of the targeted platform. This includes retrieving the available parameters and their settings.

Call the GetParamterInfos() method of CADISimulationFactory to retrieve the parameter information. It returns a list with descriptions of the configurable parameters (that is of data type CADIParameterInfo_t). This includes information such as the parameter ID for later reference, the parameter type, and the default value.

The caller can create a list of parameter values (of type CADIParameterValue_t) that are used for configuration of the platform. This list must end with an extra element that has the parameter ID 0xFFFFFFFF. It is required to add this element because not all parameters require setting and the order of the parameters within the list is undefined.

Parameters that are not part of the value list sent by the caller are set to their default value.

The ID `0xFFFFFFFF` is equal to `static_cast<uint32_t>(-1)`.
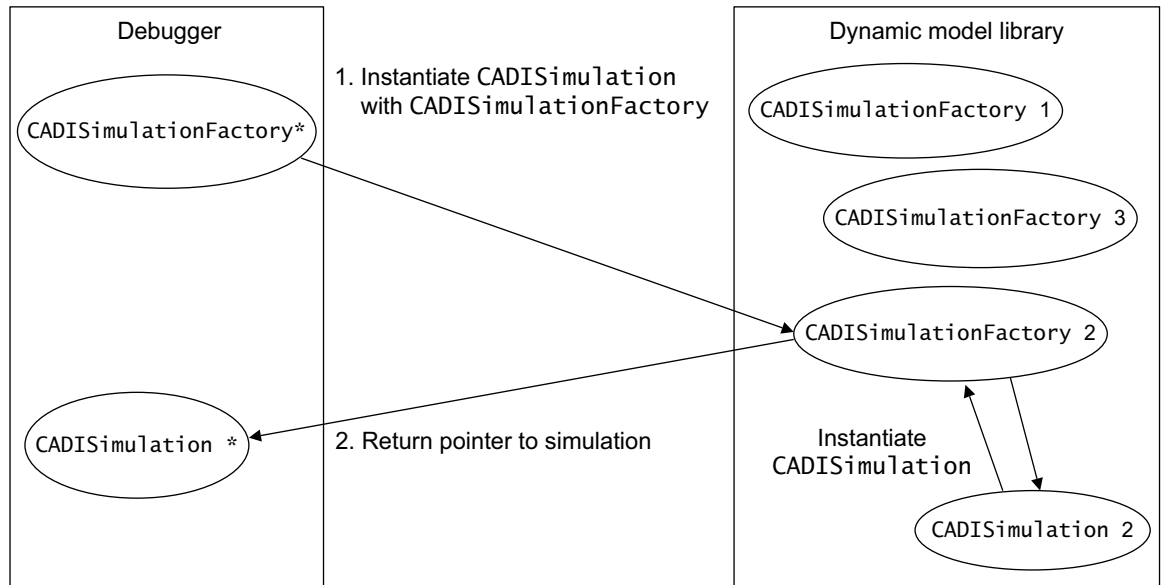
Figure 2-5 Instantiating a CADI simulation

The list of parameter values is forwarded to the `Instantiate()` method of the simulation factory. This call creates the actual CADI simulation. It might also receive a pointer to a `CADIErrorCallback` object and a pointer to a `CADISimulationCallback` object. These objects are automatically registered to the newly instantiated CADI simulation and must be provided by the caller.

The result of the simulation instantiation is a pointer to a `CADISimulation` object as shown in Figure 2-5. A compatibility check consisting of its `ObtainInterface()` method and calling `static_cast()` must be performed.

After the CADI simulation is created, the simulation factory is no longer required. The pointer to the corresponding CADISimulationFactory can therefore be released. This can be safely done for the following reasons:
* the CADI simulation is managed by the CADI broker
* the simulation factory can be retrieved again from the broker if required.

Example 2-3 shows the entire process:

Example 2-3 Getting the simulation factory

```
//having already obtained a pointer to the CADIBroker before
//which is called cadi_broker
//callback objects, will be registered to CADISimulation
MyCADIErrorCallback errorCallbackObject;
MyCADISimulationCallback simulationCallbackObject;

//enable vector for MyCADISimulationCallback
char simulationCallbacksEnable[eslapi::CADI_SIM_CB_Count];
```

```
//enable all callbacks of MyCADISimulationCallback
memset(simulationCallbacksEnable,
        1, eslapi::CADI_SIM_CB_Count * sizeof(char));


//preparing parameters for GetSimulationFactories()
uint32_t desiredNumberOfFactories = 10; //arbitrarily chosen, must be large
                                        //enough to get all factories
uint32_t startFactoryIndex = 0;
uint32_t actualNumberOfFactories = 0;


// array of CADISimulationFactory pointers to store the broker's factories
eslapi::CADISimulationFactory** factoryList =
        new eslapi::CADISimulationFactory*[desiredNumberOfFactories]();


eslapi::CADIReturn_t status;

status = cadi_broker->GetSimulationFactories(startFactoryIndex,
                        desiredNumberOfFactories,
                        factoryList,
                        &actualNumberOfFactories);
if (status != eslapi::CADI_STATUS_OK)
{
   //GetSimulationFactories() failed
}


//...decide which entry in factory list to use,
//let's assume we chose the second (index '1'!!)...
//check compatibility of factory
eslapi::if_name_t ifNameFactory = "eslapi.CADISimulationFactory2";
eslapi::if_rev_t  minRevFactory = 0;
eslapi::if_rev_t  actualRevFactory = 0;
if (factoryList[1]->ObtainInterface(ifNameFactory,
                                    minRevFactory,
                                    &actualRevFactory) == NULL)
{
     //factory is incompatible
}
//continue with instantiation of a simulation...
uint32_t desiredNumberOfParameterInfos = 20; //arbitrarily chose, must
                               //be large enough to store all parameter infos
uint32_t startParameterInfoIndex = 0;
uint32_t actualNumberOfParameterInfos = 0;
eslapi::CADIParameterInfo_t* parameterInfoList = new
eslapi::CADIParameterInfo_t[desiredNumberOfParameterInfos]();
status = factoryList[1]->GetParameterInfos(startParameterInfoIndex,
                                    desiredNumberOfParameterInfos,
                                    parameterInfoList,
                                    &actualNumberOfParameterInfos);
if (status != eslapi::CADI_STATUS_OK)
{
   //GetParameterInfos() failed
}
eslapi::CADIParameterValue_t* parameterValues =
  new eslapi::CADIParameterValue_t[actualNumberOfParameterInfos + 1]();
                                  // + additional "terminating" element
// ...fill the parameter values accordingly...
// set terminating element
parameterValues[actualNumberOfParameterInfos].parameterID =
                                    static_cast<uint32_t>(-1);
cadi_simulation = factoryList[1]->Instantiate(parameterValues,
                        &errorCallbackObject,
                        &simulationCallbackObject,
                        simulationCallbacksEnable);
```

```
if (cadi_simulation == NULL)
{
      //instantiation failed
}
//check compatibility of simulation
eslapi::if_name_t ifNameSimulation = "eslapi.CADISimulation2";
eslapi::if_rev_t  minRevSimulation = 0;
eslapi::if_rev_t  actualRevSimulation = 0;
if (cadi_simulation->ObtainInterface(ifNameSimulation,
                                     minRevSimulation,
                                     &actualRevSimulation) == NULL)
{
      //interface incompatible
}

//no longer needed
//release CADISimulationFactories, no longer needed
for (uint32_t i = 0; i < actualNumberOfFactories; i++)
      factoryList[1]->Release();

//no longer needed, destroy just the array
delete[] factoryList;

//continue with obtaining the CADI interface from simulation...
```

2.5 Getting existing CADI simulations

If the caller connects to a running CADI simulation, it must retrieve information on already existing simulations by calling the `GetSimulationInfos()` method of the CADI broker as shown in Figure 2-6. This call returns an internal list of available simulations that is maintained by the broker. The number of elements retrieved depends on:

- the size of the buffer used to fetch the list
- the number of simulations that are available
- the specified start index into the internal buffer in the broker.

The list of existent simulations held by the broker can change dynamically, so it might be necessary to update this list regularly to detect the creation or destruction of CADI simulations.
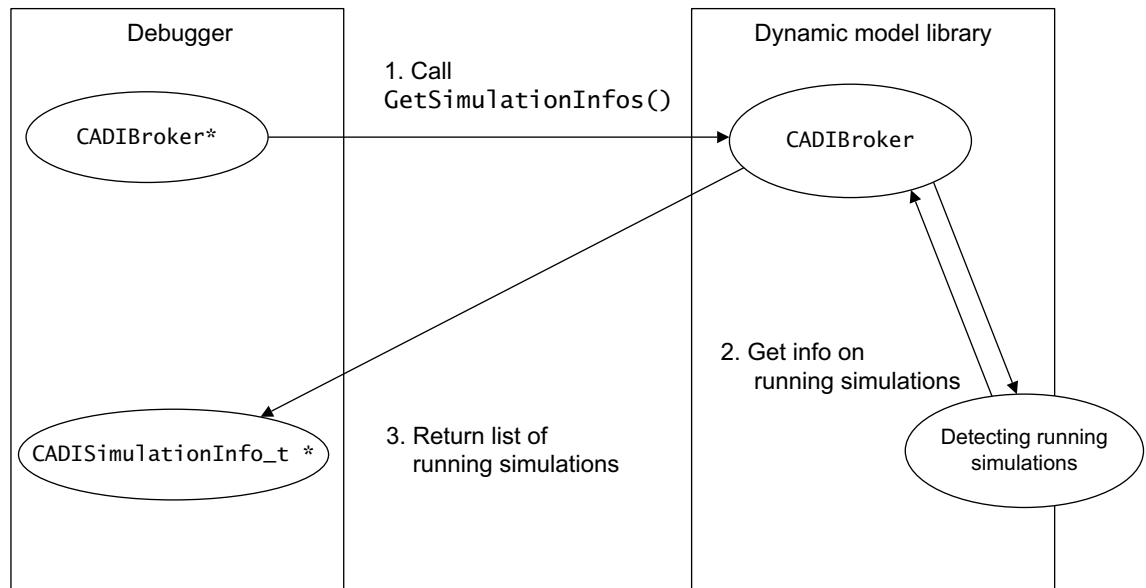
Figure 2-6 Getting information on existing CADI simulations

Based on the acquired information, the caller uses `SelectSimulation()` to select a simulation to attach to. To specify a simulation, its ID (as part of the simulation info) must be used.

`SelectSimulation()` might receive pointers to a `CADIErrorCallback` object and a `CADISimulationCallback` object. These objects are automatically registered to the requested simulation and must be provided by the caller.

——— **Note** ———

It is not required that a specific simulation ID matches the corresponding index for the simulation within the internal list held by the broker.
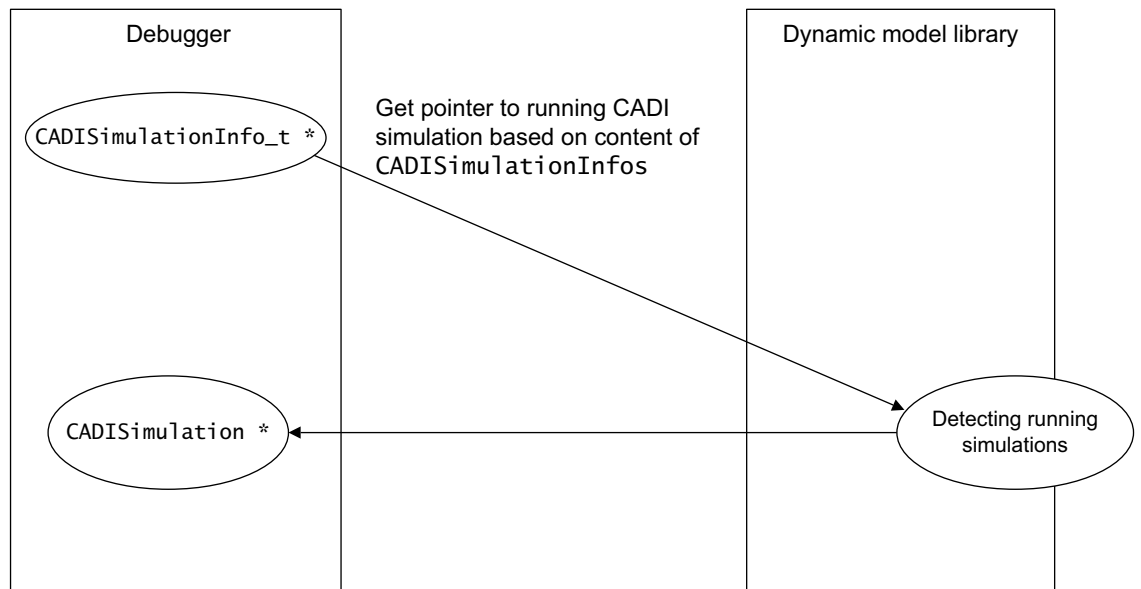
Figure 2-7 Getting an existing CADI simulation

The result of `SelectSimulation()` is a `CADISimulation` pointer to the requested simulation. The `ObtainInterface()` method and the `static_cast` scheme must be applied to check validity.

Example 2-4 shows a typical implementation:

Example 2-4 Getting an existing CADI simulation

```
//having already obtained a pointer to the CADIBroker before
//which is called cadi_broker
MyCADIErrorCallback errorCallbackObject;
MyCADISimulationCallback simulationCallbackObject;
char simulationCallbacksEnable[eslapi::CADI_SIM_CB_Count];
memset(simulationCallbacksEnable,
       1, eslapi::CADI_SIM_CB_Count * sizeof(char)); //enable all callbacks
uint32_t desiredNumberOfSimulations = 10;
uint32_t startSimulationInfoIndex = 0;
uint32_t actualNumberOfSimulations = 0;
eslapi::CADISimulationInfo_t* simulationList = new
               eslapi::CADISimulationInfo_t[desiredNumberOfSimulations]();
eslapi::CADIReturn_t status;
status = cadi_broker->GetSimulationInfos(startSimulationInfoIndex,
                                         desiredNumberOfSimulations,
                                         simulationList,
                                         &actualNumberOfSimulations);
if (status != eslapi::CADI_STATUS_OK)
{
    //GetSimulationInfos() failed
}
//... decide which simulation to connect to, for this example using the
//second one (index '\1'!!)...
CADISimulation* cadi_simulation
               = cadi_broker->SelectSimulation(simulationList[1].id,
                                               &errorCallbackObject,
                                               &simulationCallbackObject,
                                               simulationCallbacksEnable);
if (cadi_simulation == NULL)
{
```

```
        //connection to simulation failed
}
//check compatibility
eslapi::if_name_t ifNameSimulation = "eslapi.CADISimulation2";
eslapi::if_rev_t  minRevSimulation = 0;
eslapi::if_rev_t  actualRevSimulation = 0;
if (cadi_simulation->ObtainInterface(ifNameSimulation,
                                     minRevSimulation,
                                     &actualRevSimulation) == NULL)
```

——— **Note** ———

The size of `eslapi::CADISimulationInfo_t` is approximately 8kB. When allocating arrays of this object on the stack, you must take care that stack allocation limits are not exceeded.

2.6   Getting a target interface

After obtaining a `CADISimulation` pointer, an individual target can be connected to. The steps are the same for connecting to an existing simulation or for instantiating a new one.
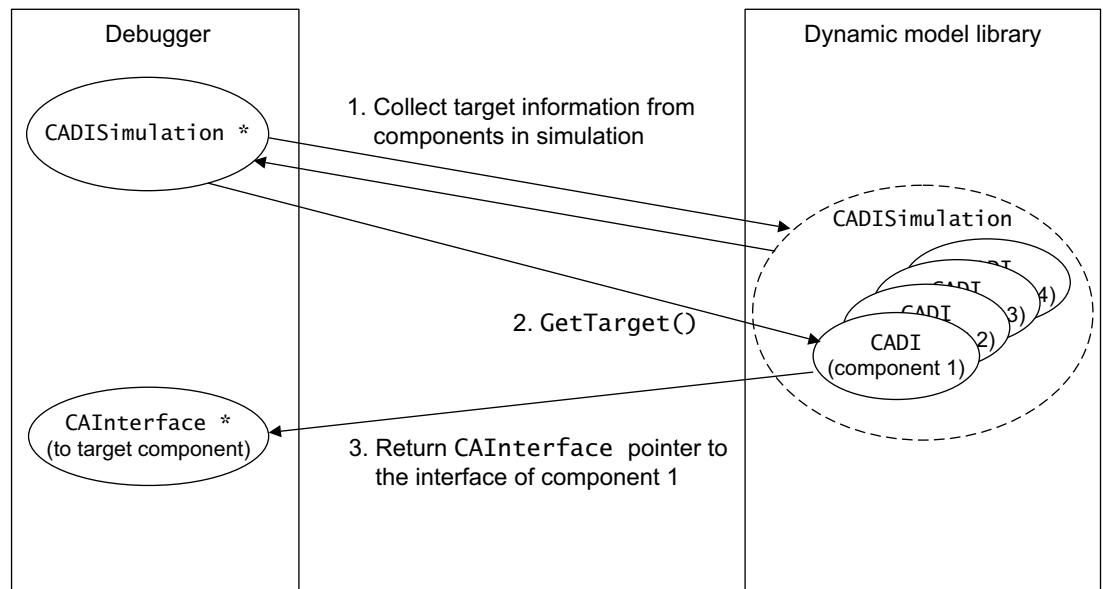
Figure 2-8 Getting a target interface

The `CADISimulation` class holds information on the contained target components that can be retrieved using the `GetTargetInfos()` method. This information includes the ID and properties of the target that might be important for a debugger such as, for example, whether the target executes software.

The caller can decide which target to connect to based on the retrieved information. The required component is specified by its ID. The ID is forwarded as a parameter to the `GetTarget()` method in a later call.

The result of the `GetTarget()` call is a `CAInterface` pointer to the implementation of the CADI interface in the target component. This pointer is then used to obtain the required interface in combination with a compatibility check by calling `ObtainInterface()`. Typically, the requested interface is of type CADI as shown in Figure 2-9 on page 2-19, but other interfaces such as `CADIDisassembler`, `CADIProfiling`, or a custom extension, can also be requested.

After acquiring another non-NULL `CAInterface` pointer, the caller must perform a `static_cast` to the appropriate type to access its full functionality.
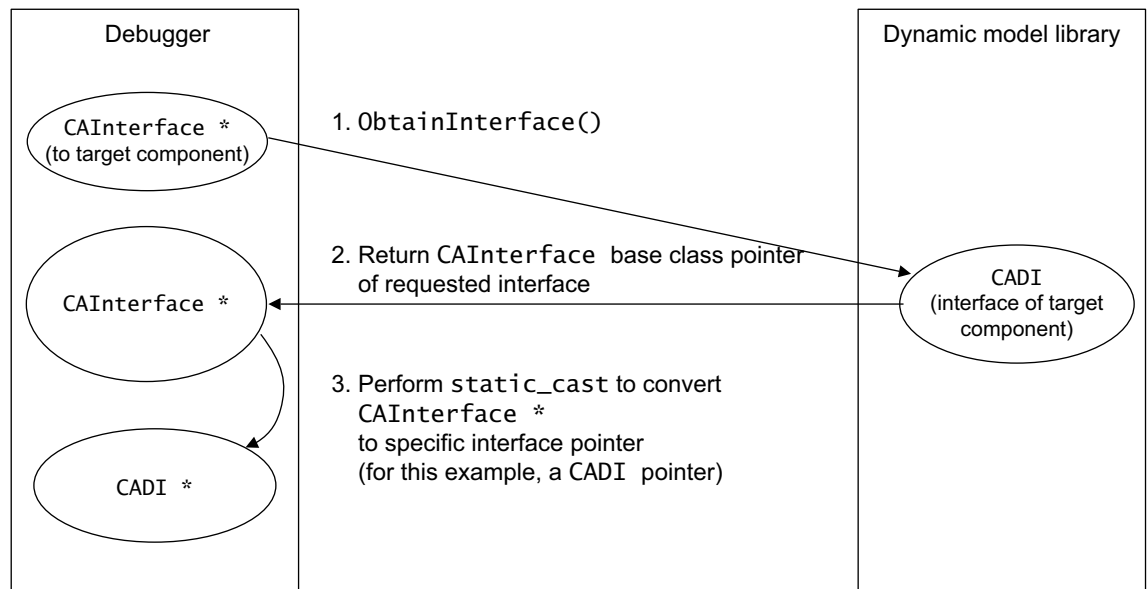
Figure 2-9 Getting a CADI interface

Example 2-5 shows a typical implementation:

Example 2-5 Getting a CADI interface

```
//having already obtained a CADISimulation pointer called
//cadi_simulation
uint32_t desiredNumberOfTargetInfos = 20; //arbitrarily chosen, must be
                                    //large enough to get all targets
uint32_t startTargetInfoIndex = 0;
uint32_t actualNumberOfTargetInfos = 0;
eslapi::CADITargetInfo_t *targetInfoList =
            new eslapi::CADITargetInfo_t[desiredNumberOfTargetInfos]();
status = cadi_simulation->GetTargetInfos(startTargetInfoIndex,
                                    desiredNumberOfTargetInfos,
                                    targetInfoList,
                                    &actualNumberOfTargetInfos);
if (status != eslapi::CADI_STATUS_OK)
{
    //GetTargetInfos() failed
// ...
// decide which target to connect to, we take the fourth (index '3'!!)
// ...

eslapi::CAInterface* ca_interface =
                    cadi_simulation->GetTarget(targetInfoList[3].id);
if (ca_interface == NULL)
{
    //GetTarget() failed
}

//requesting CADI 2.0 interface
eslapi::if_name_t ifNameTarget = "eslapi.CADI2";
eslapi::if_rev_t  minRevTarget = 0;
eslapi::if_rev_t  actualRevTarget = 0;
ca_interface = ca_interface->ObtainInterface(ifNameTarget,
                                        minRevTarget,
                                        &actualRevTarget);
```

```
if (ca_interface == NULL)
{
    //unsupported or incompatible interface
}
//converting CAInterface* to CADI*
CADI* cadi = static_cast<CADI*>(ca_interface);
//it might be necessary to connect to other targets later on, hence
//keeping the target infos for now
// continue using CADI ...
```

2.7 Disconnecting from a target

The target connection mechanism in CADI enables establishing connections to CADI targets. It is also responsible for a clean disconnection from targets and the release of a connected simulation object.

The primary way to disconnect from a simulation is to use the `Release()` method of those target-side classes that are involved in the connection mechanism. After this method is called, the caller must ensure that it does not start any additional interaction with the connection. The call performs the appropriate actions on the target-side such as:

* informing other connected callers
* if the simulation is to be shut down, destroying objects that are no longer used.

─── **Caution** ───

The caller must not explicitly destroy any target-side objects. This is the task of the target implementation and must be triggered through `Release()` calls wherever appropriate.

Using only `Release()` calls is acceptable for simple scenarios such as unique and direct connections between caller and target. For more sophisticated scenarios, however, a well-coordinated disconnection is required. The `CADISimulationCallback` class provides two callbacks that are essential for such a disconnection:

`simShutdown()`

> the simulation signals a request for a clean shutdown.

`simKilled()`   the simulation signals a hard shutdown. It was not able to be kept alive until a clean shutdown could be performed. After this call is received, the caller must ensure that it does not access the `CADISimulation` or associated `CADI` objects.

Using these callbacks in combination with the `Release()` method in the target enables establishing well-defined procedures for disconnection from a CADI simulation.

2.7.1 Deleting pointers to registered callbacks

A caller typically registers at least one callback object of type `CADICallbackObj` to a connected target. To avoid any access violations from the target after a caller has disconnected, the essential first step in disconnecting is to remove the pointers to all registered callback objects of the caller.

After removal of the callback object pointers, no additional action is required by the caller on the target because the cleanup of the `CADI` objects is managed by the underlying CADI simulation.

2.7.2 Releasing the objects of the target connection mechanism

In a simple scenario, the release of the CADI target connection mechanism is not complex. It works in the reverse order of establishing a connection:

1. The `CADISimulation` must be released for a clean disconnection. Depending on the shutdown parameter for the method, the simulation is kept alive or destroyed.

2. The `Release()` method of `CADISimulation` is responsible for initiating the clean up of the existing CADI interfaces for a simulation shutdown.

   Additionally, the call must close any other connection to the simulation by issuing the corresponding simulation callbacks. After that it is guaranteed that all connections are removed, the simulation object and all of its members can be cleanly destroyed.

3. If a CADI factory was used to instantiate a new simulation, the `CADISimulationFactory` class is next within the class hierarchy.

    As with the other CADI classes, it owns a `Release()` method but, as mentioned in *Using GetSimulationFactories()* on page 2-11, the factory can be immediately released after instantiating the required CADI simulation. It is not necessary to call `Release()` on the factory during shutdown.

4. The last step in closing a connection is to release the CADI broker. After cleanly releasing all simulations and factories owned by the broker, the `Release()` method is only required to destroy the object it belongs to.

    It some cases, however, a broker might contain live and used members. It must ensure that any connected caller is cleanly disconnected from then and that its own members are destroyed.

2.7.3 Typical shutdown scenarios

The following sections describe the typical scenarios for shutting down a simulation:

*   *Single caller and the caller initiates shutdown*
*   *Single caller and the simulation initiates shutdown*
*   *Multiple callers and one of the callers initiates shutdown* on page 2-23
*   *Multiple callers and the simulation initiates shutdown* on page 2-24.

Single caller and the caller initiates shutdown

A single connected caller initiating a simulation shutdown is the most typical scenario. The procedure consists of a `Release()` call to the simulation with either `true` or `false` as the `shutdown` parameter as shown in Figure 2-10. Depending on the parameter value, the simulation is destroyed or kept alive.
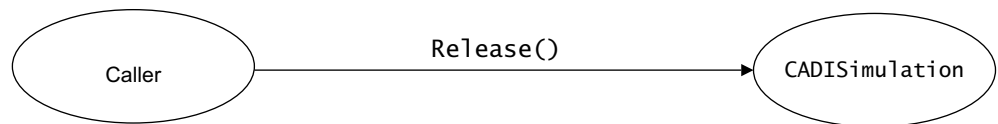
Figure 2-10 Single caller and simulation shutdown initiated by caller

Single caller and the simulation initiates shutdown

The simulation initiates its shutdown and informs the caller. This is used, for example, if the simulation offers a user-interface for interaction that permits ending the simulation. The procedure requires two steps:

1. The simulation that is shutting down, for example because of a corresponding semihosting input, issues a `simShutdown()` callback through the registered simulation callback object as shown in Figure 2-11 on page 2-23.

2. The first reaction of the attached caller is to unregister any callback object that is registered to targets owned by the simulation.

3. After unregistering the callbacks, the caller issues a `Release()` call to indicate that it does not access the simulation or targets in the future.
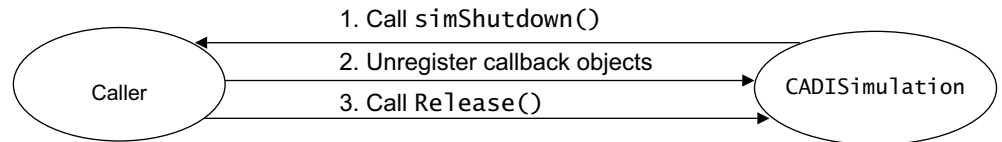
Figure 2-11 Single caller and simulation shutdown initiated by simulation

The shutdown parameter can be set to `false`, as the simulation is already shutting down. A value of `true` is ignored at this point.

Multiple callers and one of the callers initiates shutdown

Figure 2-12 shows a simulation shutdown that is initiated by a caller while multiple callers are attached.
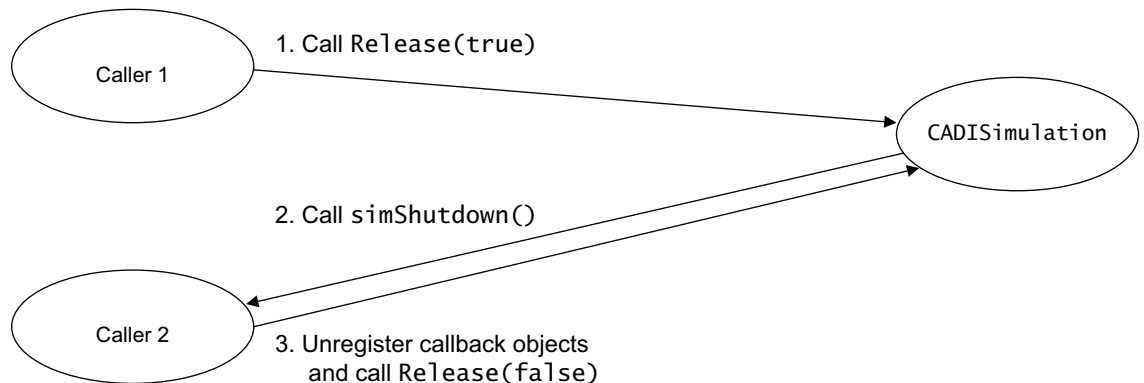
Figure 2-12 Multiple callers and simulation shutdown initiated by caller

The sequence is similar to that described in *Single caller and the caller initiates shutdown* on page 2-22 except that the other caller must also be shutdown:

1. Call the `Release()` method for the simulation. The shutdown parameter can be either `true` or `false`. If `false`, the simulation is not shut down and the sequence ends here.

2. If shutdown is `true`, there is a requirement for some interaction with all other attached callers. To indicate the demand to shut down, the simulation issues the `simShutdown()` callback to all registered simulation callback objects that are enabled for this call.

3. The informed callers must stop their communication with the simulation as soon as possible and remove any registered callback objects from the simulation and its targets.

   The affected callers must sign off with a `Release()` call to announce successful disconnection from the simulation. Its shutdown parameter is set to `false` as the shutdown is already in progress (a value of `true` is ignored at this point).

4. After all callers have disconnected from the simulation, the `CADISimulation` object can be destroyed.

5. If all callers have not disconnected, but the simulation must urgently shut down, the simulation sends a `simKilled()` callback. If this occurs, the caller must not access the corresponding simulation in the future.

Multiple callers and the simulation initiates shutdown

Multiple callers are attached to a simulation and the simulation initiates its own shutdown. This is used, for example, if the simulation offers a user-interface for interaction that permits ending the simulation. Figure 2-13 shows this scheme.
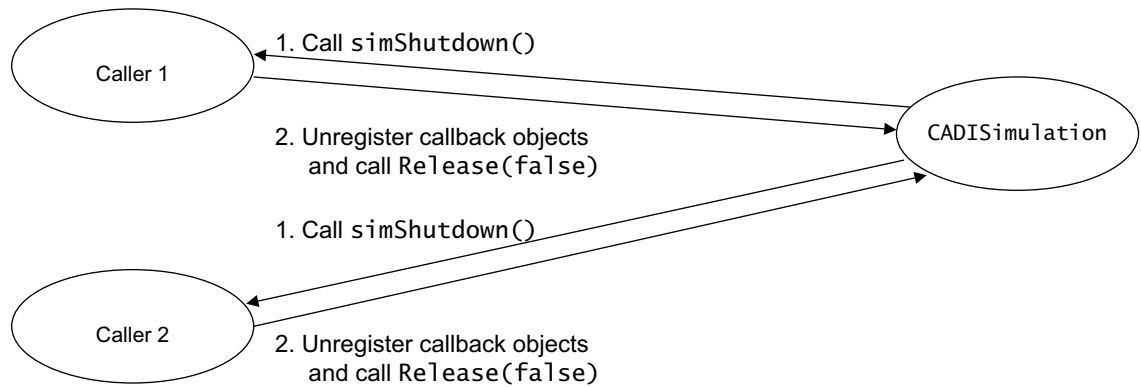
Figure 2-13 Multiple callers and simulation shutdown initiated by simulation

The main difference between this scheme and the one presented in *Multiple callers and one of the callers initiates shutdown* on page 2-23 is the missing Release(true) call:

1.   The simulation immediately issues the simShutdown() callbacks to all attached callers that have registered a simulation callback object.

2.   Each informed caller must perform a call to Release(). After all attached callers are signed off, the simulation can be safely destroyed.

Chapter 3
Using the CADI Interface Methods from a Debugger

This chapter describes how a debugger uses the CADI interface to control the target. It contains the following sections:

3.1 Overview of CADI accesses from a debugger

Using the CADI interface requires specific calling schemes and procedures:

- Some are typically used for all targets such as, for example, setting up a target connection.

- Some might be deployed for dedicated functionality such as, for example, writing to memories.

This chapter describes typical schemes and the general usage of the CADI interface from the caller side.

——— **Note** ———

Procedures that are described in separate chapters are only covered briefly in this one. See the referenced sections for more information on these topics.

A major aim of CADI 2.0 is to prevent the passing of data objects from the heap memory across dynamic library boundaries. To achieve this, each method call that passes information from the target to the caller must allocate data objects to receive the information. A pointer to this object is forwarded to the target that must fill it.

All CADI 2.0 data types provide a default constructor that initializes newly created data objects with reasonable values. This eliminates the risk that initialization is forgotten and unexpected behavior is caused by accident.

CADI 2.0 includes fundamental calling schemes for requesting hardware resource information and accessing these resources.

Methods in CADI 2.0 to request resource information typically have a prototype of the following form:

```
CADIReturn_t method_name(uint32_t startIndex,
                         uint32_t desiredNumOfElements,
                         uint32_t *actualNumOfElements,
                         dataType *buffer);
```

The following guidelines must be followed for all CADI calls:

- The `startIndex` refers to an internal list within the target that contains the requested data. If requesting information of which every element holds a specific ID, the ID does not necessarily match the list index. Consequently, IDs are not required to be sequential.

- The size of the `buffer` array must match the `desiredNumOfElements` parameter. This is necessary to guarantee enough memory for passing the requested data.

- The number of elements requested in `desiredNumOfElements` *must* always be larger or equal to the actually returned number of elements. Otherwise, the used buffer is too small and this might lead to undesired effects.

- If more data elements than available are requested, only the existing elements are returned. This results in `buffer` containing a smaller number of elements than requested. The available elements are copied into `buffer` starting from position zero. The call finishes with `CADI_STATUS_OK`.

- Even if a call fails, some data elements might have been successfully set. If so, `actualNumOfElements` must provide this number.

- If the `startIndex` points behind the last position of the internal list held by the target, the call ends successfully and returning `CADI_STATUS_OK`, but `actualNumOfElements` is zero.

Other similar schemes exist. The returned `CADIReturn_t` and the `actualNumOfElements` parameter are set accordingly.

If querying certain resource information, the expected number can be usually obtained in the form of target properties returned by previous method calls. There are, however, some methods such as `GetSimulationFactories()` and `GetSimulationInfos()` for which the caller cannot know the exact number of properties in advance. For such calls, it is necessary to estimate a reasonable number that is sufficient to receive all expected items.

If the complete array is filled for such calls, it might be necessary to repeat the call with a larger array because a completely filled array might mean both a number of items that exactly matched the requested one and a number of items that was too small. Because this case cannot be excluded, it is therefore necessary to ask for more items to assure that all items have been acquired.

3.1.1 CADI and threads

Debugging a simulation model that exposes a CADI implementation typically uses the following multiple threads:
- one simulation thread
- one (debugger) thread for each connected debugger

To decouple the threads (especially the debugger threads from the simulation thread) and avoid deadlocks, you must obey the following rules when implementing the interface:

- Methods of the classes `CADI`, `CADIDisassembler`, `CADISimulation`, `CADISimulationFactory`, `CADIBroker`, and `CADIProfiling` must only be called from a debugger thread.

- Methods and callbacks from the callback classes `CADIProfilingCallbacks`, `CADIErrorCallback`, `CADISimulationCallback`, and `CADICallbackObj` must only be called from the simulation thread.

This implicitly means that:
- a CADI callback method must never directly call a normal CADI method
- a normal CADI method must never directly call a CADI callback method.

3.2 CADIReturn_t return values

Most CADI 2.0 methods return a value of type `CADIReturn_t`. The return value:
- informs the debugger that the method call succeeded.
- gives the debugger a hint about what happened and how to proceed.

The `CADIReturn_t` object provides hints that are of value in classifying the error. The debugger can take appropriate action such as repeating a call with different parameters or triggering a fallback solution if the functionality is not supported. If required, more detailed information about a failure can be read from the target with the `CADIXfaceGetError()` method that is accessible through the `CADI` object of the target.

The possible return values are:

CADI_STATUS_OK

> The method call completed successfully. The debugger is not required to take any additional action.

CADI_STATUS_ArgNotSupported

> An argument that in principle can be processed is, however, not supported by the current target. This might be, for example, a register ID that is not assigned to any register or a memory address that does not belong to an addressable memory range.
>
> The action the debugger must take depends on the unsupported argument:
>
> - If the argument represents a certain capability of the target, for example the `stepOver` argument of `CADIExecSingleStep()`, the debugger must switch to a fallback solution.
> - If an argument such as the `groupID` of `CADIRegGetMap()` is unsupported, this might be because the debugger used the wrong information.

CADI_STATUS_IllegalArgument

> Indicates that the client issued a call that is disallowed by the CADI specification. The client must not rely on the target handling an illegal call correctly.
>
> An illegal argument also refers to values that can never be accepted by an implementation of the method. This especially applies to values that represent an invalid CADI data object or to a pointer that has not been set to a valid object. For example, calling `CADIBptClear()` for a breakpoint ID of 0 (which is reserved for invalid breakpoints) must result in this return value.
>
> Another important example of illegal arguments is the use of null pointers that are not explicitly permitted. If a CADI method returns with this value, the implementation of the corresponding debugger functionality is defective.

CADI_STATUS_CmdNotSupported

> The called method is not implemented for the addressed target. An implementation of a CADI call returning this value must never return a different one. The client can assume that all future calls to the same method also return this value.
>
> The debugger must react to this response with a fallback solution. If no fallback is available, the debugger cannot use the requested method on the selected target.

CADI_STATUS_UnknownCommand

> This value must only be returned by methods that receive a command string such as `CADIXfaceBypass()`. It must be used if an unknown command is received. It is completely up to the target which commands are known and unknown.

CADI_STATUS_TargetBusy

    The CADI call could not be completed because the target is busy. Registers and memories, for example, might not be writable while the target is executing application code. The target is typically not in a stable state and must return this value.

    The debugger can either wait for the target to reach a stable state or enforce a stable state by, for example, stopping a running target. The debugger can repeat the original call after the target reaches a stable state.

CADI_STATUS_TargetNotResponding

    The target did not respond to the call and the method timed out. This might be the result of a stalled simulation or, if debugging over a network, a lost connection.

    The debugger can attempt to determine the reason the call failed by, for example, calling CADIXfaceGetError(). Depending on the result, the debugger might try to call the target again or it might attempt to safely clean up the connection.

CADI_STATUS_GeneralError

    An error occurred that is not covered by one of the more precise return values.

    The debugger can call CADIXfaceGetError() to determine the reason the call failed. Depending on the result, the debugger might attempt to handle the error.

CADI_STATUS_PermissionDenied

    Method failed because of an access being denied, such as, for example, writing a read-only register.

    This typically indicates a wrongly-configured access of a target resource.

CADI_STATUS_SecurityViolation

    Method failed because of a security violation such as, for example, reading memory with restricted access.

    This typically indicates a wrongly-configured access of a target resource.

CADI_STATUS_BufferSize

    A character string buffer used to receive a response from a CADI method is too small to carry the entire string.

    It is dependent on the implementation in target whether an empty string is returned or if the buffer is partially filled with the message based on the length of the buffer.

    ARM recommends that the debugger does not rely on the returned information. The debugger must repeat the call using a larger string buffer.

CADI_STATUS_InsufficientResources

    The method did not complete because of missing resources such as, for example, the simulation was not able to allocate enough memory on the host machine.

    To determine which of the resources are insufficient, the debugger must call CADIXfaceGetError(). Depending on the result, the debugger might repeat the failed call with a different set of arguments or use a different call to achieve the wanted result.

3.3 Target connection and configuration

Using a CADI interface requires that you first establish a connection to the corresponding target. To establish a connection:

1. open the dynamic library that implements the CADI interface
2. establish a connection to a required simulation
3. obtain the interface of the target to debug.

A summary of the steps is provided in this section. See Chapter 2 *The Target Connection Mechanism* for a detailed description on establishing a target connection.

3.3.1 Obtaining an interface pointer to the target

Figure 3-1 on page 3-7 shows the steps required to obtain the interface pointer:

1. The caller queries the target component interface for a `CAInterface` pointer.

2. The caller (for example, a debugger) acquires a `CAInterface` pointer of the targeted component. This is typically requested from a CADI simulation.

3. The caller must call the `ObtainInterface()` method of the target and pass the required interface name and revision to check for compatibility with the required interface.

4. If the requested interface is found, another `CAInterface` pointer is returned that points to the requested interface. This might be the same as the previously acquired pointer. A `NULL` pointer is returned it there is not a matching interface.

5. The caller knows that the target provides the required interface and the `CAInterface` pointer must be converted to the proper interface class, in this case `SpecificInterface`.

   It is necessary to perform a `static_cast` at this point because the boundary of a dynamic library was crossed and this prevents the use of a `dynamic_cast`. The impossibility of using a `dynamic_cast` across dynamic library boundaries was the primary reason for introducing `ObtainInterface()` followed by the `static_cast`.
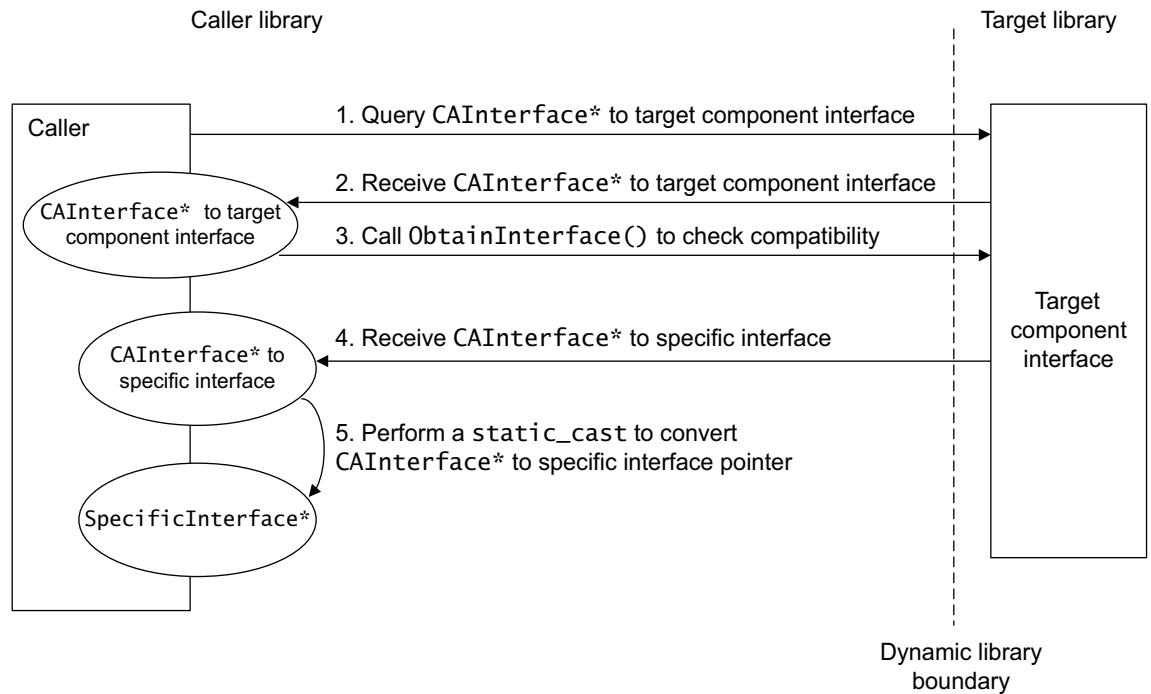
Figure 3-1 Obtaining a pointer to a specific interface

3.3.2 Target interface setup

After the CADI interface for a specific target component is obtained, there are some typical steps that must be performed to prepare the interface for the actual communication between caller and target:

The first method of the CADI interface that must be called after establishing a connection is `CADIXfaceGetFeatures()`. It returns information on the features supported by the target. These include, for example, the supported types of breakpoints, the number of register groups and memory spaces, and the register ID of the program counter. This information can be used by subsequent CADI method calls. See *CADI target characteristics* on page 3-8.

Before starting the real interaction with the connected target, the caller must register its `CADICallbackObj` objects (typically there is only one) to the corresponding CADI interface. The `CADIXfaceAddCallback()` method in the interface must therefore be called. In addition to a pointer to the callback object, an array of chars is forwarded that contains the enable vector that describes which of the callbacks in the object are permitted to be used by the target.

The enable vector that is forwarded in combination with a callback object is only associated with that specific object. You can connect different callback objects that implement different subsets of callbacks. It is also possible to re-configure a registered object by executing `CADIXfaceAddCallback()` using the same pointer in combination with a new enable vector.

3.3.3 Setting runtime parameters

CADI provides a dedicated set of method calls to set runtime parameters through the CADI interface. To retrieve information on the available parameters, the `CADIGetParameters()` method can be used. The prototype for the method uses the typical scheme receiving:

- a start index into the internal list of the target
- the required number of queried elements
- the actually returned number of elements.

An alternative way to acquire information for a single parameter is to use the `CADIGetParameterInfo()` call. It receives the name of the parameter that was, for example, determined using the list as retrieved by `CADIGetParameters()`.

After the caller has obtained parameter descriptions, the corresponding values can be queried by `CADIGetParameterValues()`. To achieve this, an array that contains the corresponding data structures must be forwarded. The elements of the array are initialized with the necessary identifiers. The size of the array is specified by the `actualNumOfParams` parameter.

Setting the runtime parameters for the target is performed in a similar manner. A list of parameters to set is created and forwarded. The `CADISetParameters()` method might return an error message that indicates the first encountered error. Based on this information, the caller can determine which parameter has caused the problem.

3.3.4   CADI target characteristics

The key characteristics for a CADI target are provided by its target features that are stored in an object of data type `CADITargetFeatures_t`. The object can be acquired by the `CADIXfaceGetFeatures()` method of the object.

`CADITargetFeatures_t` is closely related to `CADITargetInfo_t` which can be retrieved by the `GetTargetInfos()` method for a CADI simulation. The target info provides an overview of the high-level capabilities for the target such as parameterization or software execution capabilities. The target features, however, go into more detail about a specific target and inform the debugger about target resources required to configure a retargetable debugger.

The target features include:
- the number of memory spaces and register groups
- the supported breakpoint types
- the number of available reset and execution modes.

These features can help the debugger to systematically read architectural details about the target. The maximum number of returned descriptions (that is, the size of the internal lists of the target) for the associated CADI methods are equal to the corresponding number in the `CADITargetFeatures_t` struct. For example, the numbers of supported reset levels and execution modes must match the maximum number of list elements returned by `CADIExecGetResetLevels()` and `CADIExecGetModes()`.

For a single program counter, the target features denotes its register ID and enables reading it without having to search for this ID.

Extended Target Features Register

An important target feature helping a debugger to adjust to the current target is the Extended Target Features Register. After it is enabled by the corresponding flag, this string register can communicate additional features and characteristics of a target to the connected caller.

The Extended Target Features Register contains a string of tokens or arbitrary non-colon-ASCII characters separated by colons. Such a string might, for example, look like the following:

```
FOO:BAR:ANSWER=42:STARTUP=0xe000
```

The supported tokens and their semantics are implementation specific. CADI 2.0 and the Fast Models from ARM provide a predefined set of tokens that can be exposed by the target. See *CADITargetFeatures_t structure* on page B-5 for more information on these tokens.

3.3.5 Querying the hardware resource for register information

The register information is organized hierarchically as shown in Figure 3-2.
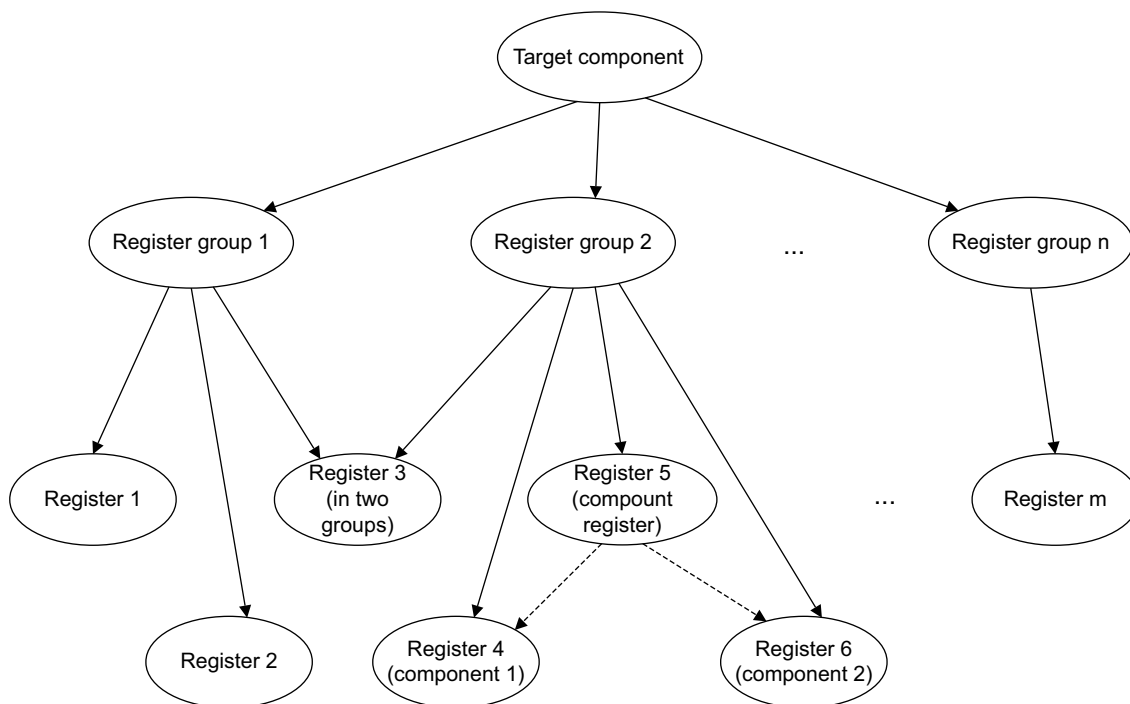
Figure 3-2 Register organization

The caller must query this hierarchy to obtain information on a specific register. The first step is to examine the information provided by the target features that contains the number of available register groups. Calling `CADIRegGetGroups()` for a specific group retrieves more detailed information. The call scheme is similar to the one presented in *Overview of CADI accesses from a debugger* on page 3-2.

Register groups are groups of registers that, for example, provide a dedicated functionality such as separating integer and floating point registers or that are used in a specific user mode. A register can be part of more than one register group.

—— **Note** ——

Register IDs *must* be unique within a target component.

After obtaining the register group information, you query the register map for a register group. by calling `CADIRegGetMap()`. In addition to the typical call scheme presented in *Overview of CADI accesses from a debugger* on page 3-2, this method receives the register group ID specified in `CADIRegGroupt_t`.

This data structure holds the number of registers that are assigned to this group. The correct size can be determined and used for the forwarded array. The result of this call is an array containing more detailed information on all registers available from this group.

To retrieve the information on all registers of a target component, the caller might iterate over all register groups and call `CADIRegGetMap()`, resulting in a concatenated list.

A register might, however, be part of more than one register group, and the resulting list might have multiple entries for the same register.

Example 3-1 shows how to access register information:

Example 3-1 Accessing register information

```
//"cadi" points to a CADI 2.0 interface

eslapi::CADITargetFeatures_t target_features;
eslapi::CADIReturn_t status;

status = cadi->CADIXfaceGetFeatures(&target_features);

//...check status and do some setup stuff...

eslapi::CADIRegGroup_t* reg_groups =
        new eslapi::CADIRegGroup_t[target_features.nrRegisterGroups]();
uint32_t groupIndex = 0;
uint32_t actualNumOfRegGroups = 0;

status = cadi->CADIRegGetGroups(groupIndex,
                                target_features.nrRegisterGroups,
                                &actualNumOfRegGroups,
                                reg_groups);
//...check status...

for (uint32_t regCnt = 0; regCnt < actualNumOfRegGroups; regCnt++)
{
   uint32_t startRegisterIndex = 0;
   uint32_t desiredNumOfRegisters = reg_groups[regCnt].numRegsInGroup;
   uint32_t actualNumOfRegisters = 0;

   eslapi::CADIRegInfo_t* reg =
                  new eslapi::CADIRegInfo_t[desiredNumOfRegisters]();

   status = cadi->CADIRegGetMap(reg_groups[regCnt].groupID,
                                startRegisterIndex, desiredNumOfRegisters,
                                &actualNumOfRegisters, reg);

   //...check status and use the obtained register information...

   delete[] reg;
}

delete[] reg_groups;

//...
```

An alternative, and much more convenient, way to obtain all register information is to call
`CADIRegGetMap()` with `CADI_REG_ALLGROUPS` as register group ID. This alternative also eliminates
redundant entries,

To allocate an array of an appropriate size, the caller can either roughly estimate the required
number or sum up the number of registers for each register group. This must result in an array
that is larger than, if there are multiple entries, or equal to the required size as shown in
Example 3-2 on page 3-11:

Example 3-2 Alternative method to obtain register information

```
//...
eslapi::CADIReturn_t status;
eslapi::CADIRegGroup_t* reg_groups =
        new eslapi::CADIRegGroup_t[target_features.nrRegisterGroups]();
uint32_t groupIndex = 0;
uint32_t actualNumOfRegGroups = 0;

status = cadi->CADIRegGetGroups(groupIndex, target_features.nrRegisterGroups,
                                &actualNumOfRegGroups, reg_groups);

//...check status...

uint32_t startRegisterIndex = 0;
uint32_t actualNumOfRegisters = 0;
uint32_t numOfAllRegisters = 0;
for (uint32_t regCnt = 0; regCnt < actualNumOfRegGroups; regCnt++)
{
   //sum up the numbers of registers in the register groups
   numOfAllRegisters += reg_groups[regCnt].numRegsInGroup;
}

//allocated array is large enough for all registers
eslapi::CADIRegInfo_t* all_registers =
                       new eslapi::CADIRegInfo_t[numOfAllRegisters]();

status = cadi->CADIRegGetMap(eslapi::CADI_REG_ALLGROUPS, startRegisterIndex,
                             numOfAllRegisters, &actualNumOfAllRegisters,
                             all_registers);

//...check status and do something with all_registers...

delete[] all_registers;
delete[] reg_groups;

//...
```

CADI supports compound registers. These are registers that are composed of several other registers (a 32-bit integer register might be for example composed of two 16-bit integer registers whose interpretation depends on the configuration of the processor).

A compound register is treated like any other register of the CADI interface. It can be directly used to read or write contents. It is also possible to manipulate an individual register in a compound register. See Example 3-3 for how to use the CADIRegGetCompound() method to retrieve a list with the IDs for the component registers. It applies the typical query scheme and receives the compound registers ID as an additional parameter.

——— **Note** ———

The number of components in a compound register is accessible through a union in CADIRegDetails_t data object of a CADIRegInfo_t

Example 3-3 Determining the number of compound registers

```
// cadi is a pointer to a cadi 2.0 interface
// registerInfos is an array of CADIRegInfo_t of length actualNumOfRegisters,
// obtained from a call to CADI::CADIRegGetMap()
```

```
for(uint32_t i=0; i < actualNumOfRegisters; i++)
{
    if (registerInfos[i].details.type == eslapi::CADI_REGTYPE_Compound)
    {
        uint32_t desiredNumOfComponents;
        desiredNumOfComponents = (uint32_t)registerInfos[i].details.u.compound.count;
        uint32_t actualNumOfComponents = 0;

        uint32_t *components = new uint32_t[desiredNumOfComponents]();

        cadi->CADIRegGetCompound(registerInfos[i].regNumber,0,desiredNumOfComponents,
                                 &actualNumOfComponents, components);
        for (uint32_t j = 0; j < actualNumOfComponents; j++)
        {
            // Do something with components
        }
    }
}
```

——— **Note** ———

A set of registers must not form a cyclic graph. A compound register must not be the parent of another compound register that directly or implicitly points back to the parent.

3.3.6   Querying the hardware resource for memory information

Similar to register information, memory information is also structured hierarchically. An example hierarchy is shown in Figure 3-3:
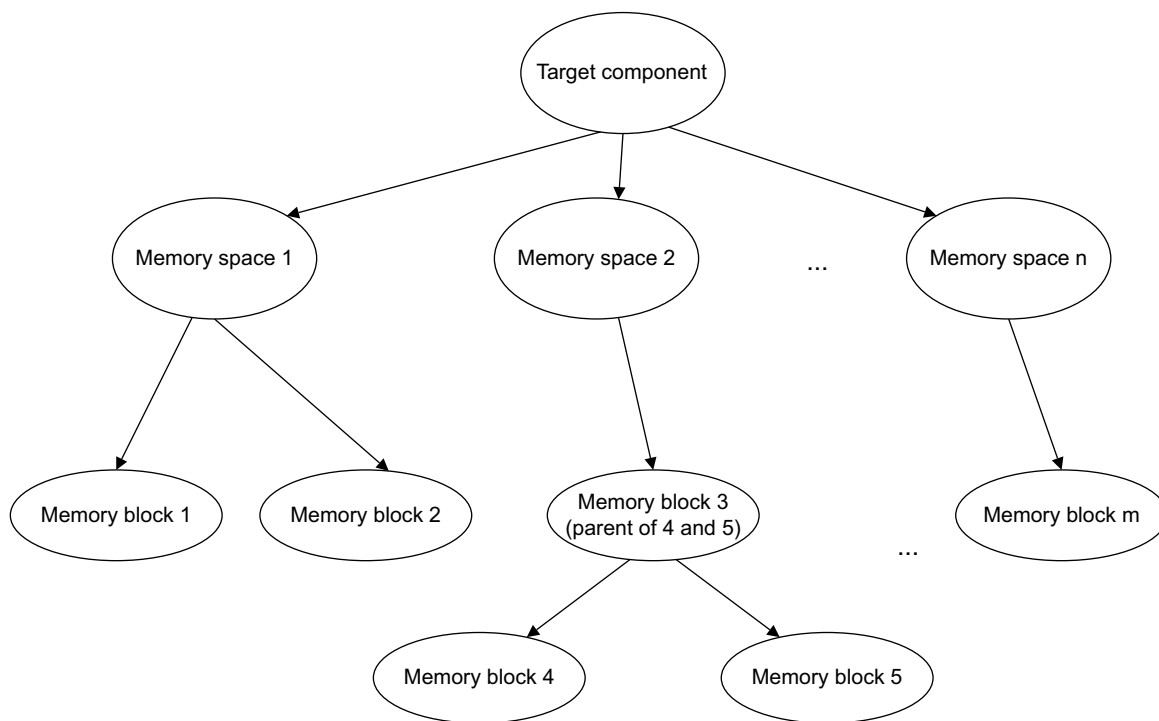
Figure 3-3 Memory organization

To retrieve the information on the memories, the caller again must start from the target features. This data structure holds the number of available memory spaces (nrMemSpaces). Based on this value, an appropriate array of CADIMemSpaceInfo_t can be created that receives the corresponding memory space information during the CADIMemGetSpaces() method call. This call complies with the common call scheme described in *Overview of CADI accesses from a debugger* on page 3-2.

A memory space is subdivided into memory blocks that define the characteristics of certain ranges of memory within a memory space such as, for example, ranges with different accessibility properties. Call CADIMemGetBlocks() to retrieve a list of these memory blocks. In addition to the parameters of the typical call scheme, it receives the memory space ID. It is not possible to acquire all available memory blocks of all memory spaces by a special memory space ID.

Memory blocks can be ordered hierarchically as shown in Figure 3-3 on page 3-12. To enable identifying the structure, the dedicated parentID parameter CADIMemBlockInfo_t is used. It is required because the memory blocks are returned as a list that flattens the corresponding hierarchy. This value must be set to the ID of the actual parent. For blocks that are direct children of a memory space, this parameter is set to CADI_MEMBLOCK_ROOT.

Example 3-4 shows how to access the memory-related hardware information:

Example 3-4 Accessing memory-related hardware information

```
//"cadi" points to a CADI 2.0 interface

eslapi::CADITargetFeatures_t target_features;
eslapi::CADIReturn_t status;

status = cadi->CADIXfaceGetFeatures(&target_features);

//...check status and setup ...
eslapi::CADIMemSpaceInfo_t* mem_spaces =
        new eslapi::CADIMemSpaceInfo_t[target_features.nrMemSpaces]();
uint32_t startMemSpaceIndex = 0;
uint32_t actualNumOfMemSpaces = 0;

status = cadi->CADIMemGetSpaces(startMemSpaceIndex, target_features.nrMemSpaces,
                                &actualNumOfMemSpaces, mem_spaces);

//...check status...
for (uint32_t spaceCnt = 0; spaceCnt < actualNumOfMemSpaces; spaceCnt++)
{
   uint32_t memBlockIndex = 0;
   uint32_t desiredNumOfMemBlocks = mem_spaces[spaceCnt].nrMemBlocks;
   uint32_t actualNumOfMemBlocks = 0;

   eslapi::CADIMemBlockInfo_t* mem_blocks =
              new eslapi::CADIMemBlockInfo_t[desiredNumOfMemBlocks]();

   status = cadi->CADIMemGetBlocks(mem_spaces[spaceCnt].memSpaceId,
                                   memBlockIndex, desiredNumOfMemBlocks,
                                   &actualNumOfMemBlocks, mem_blocks);

//...check status and use obtained memory information...
   delete[] mem_blocks;
}
delete[] mem_spaces;
//...
```

3.4   Accessing registers

This section describes how to access registers in the target.

CADIRegRead() and CADIRegWrite() are used to access registers and process an array of accesses with elements of type CADIReg_t. The elements of the array:

- specifies the addressed register by its register number (ID)

- provides a buffer of 16 bytes for accesses

- receives information about permitted access (read, write or read-write)

- optionally specifies an offset for registers wider than 128 bits. As CADIReg_t data buffer can contain a maximum of only 16 bytes, which is 128 bits. Such registers must be accessed multiple times to return all of the register content. Each access uses an appropriate offset to specify a different bit range in the register.

- enables the target to indicate registers with undefined content.

Example 3-5 shows one way to implement a read access to a register with a width of 512 bits:

Example 3-5 Accessing registers in the target

```
//"register_info" is a CADIRegInfo_t representing a register with a
//bitwidth of 512 bits, reading and displaying the register's contents;
//"cadi" is a pointer to a CADI object

uint32_t regCount = (register_info.bitsWide + 127)/128;
uint32_t regWidthInBytes = (register_info.bitsWide + 7)/8;
eslapi::CADIReg_t* reg = new eslapi::CADIReg_t[regCount]();

for (uint32_t i = 0; i < regCount; i++)
{
   reg[i].regNumber = register_info.regNumber;
   reg[i].offset128 = i;
   reg[i].isUndefined = false;
   reg[i].attribute = register_info.attribute;
   memset(reg[i].bytes, 0, sizeof(uint8_t) * 16);
}

uint32_t numOfRegsWritten = 0;
eslapi::CADIReturn_t status =
      cadi->CADIRegRead(regCount, reg, &numOfRegsWritten,
                        0 /* no side effects */);

//...check status...

if (numOfRegsWritten > 0)
{
   printf("0x");
}

//start with the most significant bits to bring it in a
//readable form
for (uint32_t i = 0; i < numOfRegsWritten; i++)
{
   uint8_t currentBuffer = reg[numOfRegsWritten - 1 - i].bytes;
   uint32_t bytesInBuffer =
                    regWidthInBytes - ((numOfRegsWritten - 1 - i) * 16);
   if (bytesInBuffer >= 16)
      bytesInBuffer = 16;
```

```
    for (uint32_t j = bytesInBuffer; j > 0; j--)
    {
        printf("%02x", currentBuffer[j-1]);
    }
}

delete[] reg;
```

In addition to the forwarded array of `CADIReg_t` data objects, the number of requested accesses is passed as `regCount`. The number of successful register accesses is returned in the `numRegsRead` (or `numRegWritten`) parameter.

---— **Note** ———

The contents of the `CADIReg_t` data buffer must be accessed in little endian, even if the target uses a different endianess. That is, the element with the smallest index of the buffer array contains the least significant byte (LSB). This implicitly means that the access with offset 0, for registers wider than 128 bytes, addresses the 16 LSBs.

The caller sets the `doSideEffects` parameter to specify whether the target must perform side effects associated with the access:

• If `true`, the target must do all side effects as usual.

• If `false`, the target must decide which side effects are inevitable and must always be performed. Other side effects are not performed.

`CADIRegRead()` might have a side effect for a clear-on-read. Typically, a target must omit all side effects for a read access if the `doSideEffects` parameter is set to `false`. This corresponds to a debug read that must not interfere with the execution of the target.

A possible side effect for a write access to a register by `CADIRegWrite()` would be triggering an interrupt. For a write access, the target can decide which side effects to perform. It might be for example necessary to change the mode of a processor according to the contents of a register even if `doSideEffects` is set to `false`.

3.4.1   Accessing string registers

Accessing string registers works slightly differently than accessing an integer or a floating-point register. In contrast to other types of registers, a string register does not own a bitwidth. The actual size of the string that is accessed through it is determined by the string itself. The bytes of the data buffer in `CADIReg_t` are read sequentially until the terminating '\0' character is reached. For a string longer than 16 bytes (including the terminating character), the `offset128` parameter must be increased and the register must be read after every set of 16 bytes.

Example 3-6 shows an example implementation:

Example 3-6 Accessing string registers

```
//"register_info" contains information on a string register

eslapi::CADIReg_t stringReg; //only one CADIReg_t required
eslapi::CADIReturn_t status;

if (register_info.display == eslapi::CADI_REGTYPE_STRING)
{
    std::string readString = "";
```

```
//set up "stringReg"
stringReg.regNumber = register_info.regNumber;
stringReg.offset128 = 0;
stringReg.isUndefined = false;
stringReg.attribute = register_info.attribute;

bool stringFinished = false;
while (!stringFinished)
{
    uint32_t numOfRegsRead = 0;
    memset(stringReg.bytes, 0, sizeof(uint8_t) * 16); //init buffer

    status = cadi->CADIRegRead(1, //regCount
                               &stringReg,
                               &numOfRegsRead,
                               0); //do no side effects

    //...check status and number of actually read registers...

    for (uint32_t i = 0; i < 16; i++)
    {
        char currentChar = stringReg.bytes[i];
        readString.append(1, currentChar);
        if (currentChar == '\0')   //reached end of string, leaving loop
        {
            stringFinished = true;
            break;
        }
    }

    stringReg.offset128++; //increment offset for next read
}
}
```

3.5 Accessing memory

Memory accesses are performed by the CADI methods `CADIMemRead()` and `CADIMemWrite()`. In contrast to register accesses, a memory access is not described by a data structure but by several parameters that must be passed to the methods.

The prototype of `CADIMemRead()`, for example, is:

```
CADIReturn_t CADIMemRead( CADIAddrComplete_t startAddress,
                          uint32_t unitsToRead,
                          uint32_t unitSizeInBytes,
                          uint8_t *data,
                          uint32_t *actualNumOfUnitsRead,
                          uint8_t doSideEffects);
```

The start address is specified in the `location.add` data member of an object of type `CADIAddrComplete_t`.

The `unitsToRead` and `unitSizeInBytes` parameters specify the number and the size of units that are accessed. The size of a unit is specified in bytes and must be a supported multiple of the *Minimum Access Size* (MAU). A list of the supported multiples can be obtained from the corresponding memory block information.

——— **Note** ———

Memory accesses must consider *invariance*. The `unitSizeInBytes` memory space property specifies the number of bytes that are treated as one unit. The coherence of these bytes is preserved, especially if converting endianess.

———

The total memory accessed in bytes is equal to the number of access units times their size in bytes. The `data` buffer used to perform the memory access is an array of `uint8_t` that must have exactly the same size as the complete access size.

The number of actually read or written access units is returned. If the memory access is completely successful, the value identified by `actualNumOfUnitsRead` value equals the number of units requested in `unitsToRead`.

——— **Note** ———

The requested number of units is not the size in bytes.

———

If an access succeeds partially, the returned number equals the one of completed units and the contents of `data` is valid for additional processing. An example for such a situation is the attempt to access memory that is not part of a memory block. This might happen if performing an access that exceeds a valid memory range.

As with data buffers for register accesses, data buffers for memory accesses are always used with little endian format.

Memory accesses can be optionally performed depending on the corresponding parameter passed to `CADIMemRead()` or `CADIMemWrite()`. As for register accesses, the target ultimately must decide which side effects can be omitted.

For `CADIMemRead()`, an example of a side effect is clear-on-read. If a read is done with the `doSideEffects` parameter set to `false`, all side effects must be omitted. Such a debug read can not interfere with the execution of the target.

A side effect during writing a memory might be, for example, the usage of a memory-mapped register whose contents controls the mode of a certain component. If this value is changed, the component must perform this side effect even if doSideEffects is set to false. If the side effect was not done, the simulated target would behave incorrectly. Example 3-7 shows write accesses to memory:

Example 3-7 Writing to memory

```
eslapi::CADI* cadi;
eslapi::CADIMemSpaceInfo_t mem_space;
eslapi::CADIMemBlockInfo_t mem_block;

//...fill the above declared variables with feasible data...

//preparing a write access to the beginning of the memory block

eslapi::CADIAddrComplete_t startAddress;
startAddress.location.space = mem_space.memSpaceId;
startAddress.location.addr = mem_block.startAddr;

//writing 256 words of 4 byte
uint32_t unitsToWrite = 256;
uint32_t unitSizeInBytes = 4;
uint32_t actualNumOfUnitsWritten = 0;

uint32_t completeAccessInBytes = unitsToWrite * unitSizeInBytes;
uint8_t* data = new uint8_t[completeAccessInBytes]();

//...filling data buffer "data"...
eslapi::CADIReturn_t status;
status = cadi->CADIMemWrite(startAddress, unitsToWrite, unitSizeInBytes,
                            data,&actualNumOfUnitsWritten, 0);
//do no side effects
//...check status and actualNumOfUnitsWritten...
delete[] data;
```

3.6 Controlling execution

This section discusses CADI features related to interactive debugging from the caller-side. This includes the management of breakpoints, the control of a targeted system, and the expected behavior of the callback methods implemented by the caller.

3.6.1 Breakpoints

Breakpoints are an essential part of any debug mechanism. CADI offers several types of breakpoints that target different areas and levels of debugging. Each breakpoint can be individually configured to modify its behavior.

CADI provides the following predefined breakpoint types:

Program Breakpoints

Program breakpoints are breakpoints set in a program memory of the target. As soon as the program counter equals hits the corresponding address, the simulation suspends and awaits additional commands from the caller.

Memory Breakpoints

A memory breakpoint can be set to a specific address in the available memory. This breakpoint suspends simulation if the specified address is read or written, or the value changes.

Register Breakpoints

Setting a register breakpoint to a specific register results in a suspended simulation if the register is read or written, or its value changes.

Instruction Step Breakpoints

The instruction step breakpoint is an inverted program breakpoint. It suspends simulation as soon as the program counter is set to an address different from the selected breakpoint address. As indicated by its name, this type of breakpoint is used for instruction step implementations. The breakpoint can be set to the current value of the program counter.

Program Range Breakpoints

This breakpoint type extends the program breakpoint to check a specific range of program addresses instead of a single one.

Exception Breakpoints

An exception breakpoint is triggered immediately after the occurrence of an exception.

The breakpoint types supported by a target component are stored in a vector that contains the features for the target (`CADITargetFeatures_t`). CADI provides comparison values to identify supported predefined types. These are named `CADI_TARGET_FEATURE_BPT_`*TypeExtension*. To determine support, perform a simple bitwise AND operation on the target features and the comparison value.

——— **Note** ———

There are two different enumeration data types that might be confused:

- `CADI_BPT_`*TypeExtension* represents an index of the breakpoint type

- `CADI_TARGET_FEATURE_BPT_`*TypeExtension* represents a breakpoint type vector for comparison with the CADI target features.

For both enumeration data types, *TypeExtension* is one of the following:

- PROGRAM
- MEMORY
- REGISTER
- INST_STEP
- PROGRAM_RANGE
- EXCEPTION
- USER_DEFINED.

Breakpoint properties

CADIBptRequest_t owns several fields specific to certain breakpoint types. These fields are ignored for other types. The following table gives an overview of the respective associations between fields in CADIBptRequest_t and the various breakpoint types.

Table 3-1 Properties for each breakpoint by trigger type

triggerType	Program	Memory	Register	Instruction Step	Program Range	Exception
Address	Yes	Yes	-	Yes	Yes	Yes
sizeOfAddressRange	-	-	-	-	Yes	-
Enabled	Yes	Yes	Yes	Yes	Yes	Yes
Conditions	Yes[a]	Yes[a]	Yes[a]	Yes[a]	Yes[a]	Yes[a]
useFormalConditions	Yes	Yes	Yes	Yes	Yes	Yes
formalCondition	Yes[b]	Yes[b]	Yes[b]	Yes[b]	Yes[b]	Yes[b]
type	Yes	Yes	Yes	Yes	Yes	Yes
regNumber	-	-	Yes	-	-	-
temporary	Yes	Yes	Yes	Yes	Yes	Yes
continueExecution	Yes	Yes	Yes	Yes	Yes	Yes

a. Yes only if useFormalConditions is 0.
b. Yes only if useFormalConditions is 1.

If a field is not supported for the required breakpoint type, its value must be left to the initial value assigned by the standard constructor of CADIBptRequest_t.

Breakpoint configuration

CADI provides the dedicated data structure CADIBptRequest_t that is used to set a breakpoint requested by the caller. It holds a description of the breakpoint and specifies its details including, for example:

- its type
- the location (memory address or register number) it is to be set to
- a possible condition for the breakpoint.

A breakpoint can be defined as enabled or as disabled and the state can be changed by a corresponding method call. Breakpoints can be configured to continue execution after being hit.

A breakpoint can be declared as temporary. Temporary breakpoints can be easily cleared by calling CADIBptClear() with a special breakpoint ID (CADI_BPT_CLEAR_ALL_TEMPORARY_BPTS). This removes all of the breakpoints that have the temporary field has set in CADIBptRequest_t.

It is not required to set every field of the corresponding data structure for a breakpoint. Properties that are not required for a certain breakpoint type are ignored by the target. For example, the triggerType field of CADIBptRequest_t is only used for setting a register breakpoint or a memory breakpoint. See *Breakpoint properties* on page 3-20.

Configuring conditional breakpoints requires special planning. There are two options, either:

•	Use the format set of conditions provided by CADI that cover typical conditions.

•	Forward the breakpoint to the target which then decides if custom conditions apply.

Using formal conditions requires that the corresponding data object owned by CADIBptRequest_t is set. This member, of type CADIBptCondition_t, includes the condition operator and a value to apply the operator to. The format of this value is described by the operator, for example whether it is a signed or unsigned value, and by the bitwidth specified in the condition data type. The bitwidth includes the sign bit.

Managing breakpoints

To set a new breakpoint, call CADIBptSet(). It receives a breakpoint description of type CADIBptRequest_t as described in *Breakpoint configuration* on page 3-20. On return, the caller receives a breakpoint ID of type CADIBptNumber_t to use in subsequent breakpoint management calls.

After creating a new breakpoint/watchpoint with CADIBptSet(), the breakpoint/watchpoint is enabled/disabled depending on the value of the **Enabled** field.

Use CADIBptConfigure() to change the enable state for a breakpoint. Call CADIBptClear() to clear a breakpoint. After clearing a breakpoint, the corresponding breakpoint number must not be referred to.

——— **Note** ———

There are two breakpoint IDs that must not be used:
•	0 represents an invalid breakpoint ID
•	0xFFFFFFFF is reserved for clearing temporary breakpoints.

To read out descriptions of currently set breakpoints either:
•	Use CADIBptRead() to request the description of a single breakpoint.
	The breakpoint number must be available to identify the required breakpoint.
•	Use CADIBptGetList() to request a list of breakpoints set in the target.
	The method can be used, for example, to read out all breakpoint information of an existent simulation the caller is connected to. No specific knowledge about the target is required.

The CADIBptGetList() method call scheme is described in *Overview of CADI accesses from a debugger* on page 3-2. To create a buffer with an appropriate size, either:

•	make a reasonable estimate of the number of breakpoints required

•	use the number of supported breakpoints specified in the target features (nrBreakpointsAvailable).

	Depending on the target implementation, this number might be very large.

An important use case for `CADIBptGetList()` is breakpoint synchronization of several connected callers. This debugger can regularly update the breakpoint list and show breakpoints that have been set from another tool.

3.6.2 Controlling the execution mode

To provide fully controlled debugging of the target, the attached debugger must be able to control the execution of the target. This capability is provided in CADI by a set of method calls that can determine the current state of the target and initiate state changes such as stopping or running. This target execution control is closely coupled to `CADICallbackObj`.

The mode, that is, the state of the target, can be explicitly requested by `CADIExecGetMode()`. This might be useful to, for example, connect to an existing simulation.

Polling of the target state is not, however, the recommended general solution. The `modeChange()` callback of `CADICallbackObj` must be implemented by the caller to eliminate the requirement for such polling calls and prevent blocking the interface. The returned mode is of type `CADI_EXECMODE_t`. The returned state is either `CADI_EXECMODE_Run`, `CADI_EXECMODE_Stop`, `CADI_EXECMODE_Error`, or `CADI_EXECMODE_ResetDone`.

— **Note** —

`CADI_EXECMODE_Bpt` cannot be returned as the target state from `CADIExecGetMode()`.

`CADIExecSetMode()` is the counterpart to `CADIExecGetMode()`. It receives a 32-bit unsigned integer as parameter. The provided value is typically of type `CADI_EXECMODE_t` which is a 32-bit unsigned integer. The intended use is to pass either `CADI_EXECMODE_Run` or `CADI_EXECMODE_Stop` to the target.

— **Note** —

`CADIExecSetMode()` cannot be used to set the target state to `CADI_EXECMODE_Bpt`.

An example of modifying the target mode is shown in Example 3-8:

Example 3-8 Accessing registers in the target

```
// very basic example of debugger accessing registers in connected target
// cadi is a connected simulation object of type CADI

cout << "Client: Invoking target->CADIExecSetMode(3)" << endl;
cadi->CADIExecSetMode(3);

cout << "Client: Invoking target->CADIExecGetMode()" << endl;
uint32_t execMode;
cadi->CADIExecGetMode(&execMode);
cout << "Client: Target's current mode is: " << execMode << endl;
```

Starting and stopping the target

For a subset of execution modes, the following dedicated methods are preferable:
- `CADIExecContinue()` instead of `CADIExecSetMode(CADI_EXECMODE_Run)`
- `CADIExecStop()` instead of `CADIExecSetMode(CADI_EXECMODE_Stop)`.

Call `CADIExecContinue()` to start or continue the execution of a target component. This asynchronous call immediately returns after triggering the target, so the execution might not start immediately. The registered callback object (from the caller) is responsible for indicating the actual beginning of the target execution by issuing a `modeChange()` callback.

If `CADIExecContinue()` is called and the target is running (`CADI_EXECMODE_Run`), the target must ignore the call and return `CADI_STATUS_TargetBusy`.

Call `CADIExecStop()` to stop a running simulation. This method returns immediately and the target is not typically stopped when the call returns. The caller must wait for a `modeChange()` callback that indicates `CADI_EXECMODE_Stop`.

If `CADIExecStop()` is called and the target is already stopped (`CADI_EXECMODE_Stop`), the call must be ignored by the target and return `CADI_STATUS_TargetBusy`.

——— **Note** ———

In general, a clients must expect that mode changes can occur asynchronously. If for example an asynchronous mode change occurred during the execution of:

```
if (t->CADIExecGetMode()==CADI_EXECMODE_Stop) t->CADIExecContinue()
```

the second call might return `CADI_STATUS_TargetBusy` while the client receives a `modeChange` message on the callback thread. The client must handle all possible outcomes of this race condition.

Stepping the target

In addition to the ability to run the target until the next breakpoint or the end of simulation, you can use `CADIExecSingleStep()` to step the target component for one or more steps.

Target steps can be specified as either cycle steps or instruction steps. That is, the target is either stepped for a specific number of clock cycles or stepped until the corresponding instructions are completely finished.

The `stepOver` parameter of `CADIExecSingleStep()` enables stepping over call instructions. This is primarily intended for use with source level debugging where some methods or function calls must not be stepped through.

The method is asynchronous and the call returns immediately and typically before the instructions have been finished. A sequence of `modeChanges()` to `CADI_EXECMODE_Run` and `CADI_EXECMODE_Stop` are issued to inform the caller about the progress of the execution.

If `CADIExecSingleStep()` is called and the target is running, the call must be ignored and `CADI_STATUS_TargetBusy` returned.

Using CADI resets

A CADI reset is intended to bring a simulation platform, or one of its components, back into a specific state. This simulation reset must be distinguished from a real hardware reset because it might perform, for example, certain initialization steps that real hardware does not do.

CADI resets are identified by their reset level and a name. The corresponding reset level numbers must be used uniquely within a target. There must not be two different resets defined to be of the same reset level.

CADI permits free definition of its simulation reset levels. Each associated reset can differ in the addressed components or resources. One reset might, for example, only initialize the core registers in a processor, but another reset might modify both the core registers and memory in the target.

CADI reserves reset 0 as a *Hard Reset* and explicitly specifies the semantics of this reset. All other reset levels, however, can be customized and might differ from model to model. Reset level numbers can be chosen arbitrarily and have no other meaning than representing a certain simulation reset. There is, for example, no ordering of reset levels by their severity.

Because CADI reset 0, the *Hard Reset*, has fixed semantics, it must be implemented by every model providing a CADI implementation. This *Hard Reset* resets all state variables of a model including those that would not be modified by a real hardware reset. After the reset, the simulation platform must be in the same state as it was immediately after instantiating it. The corresponding initialization values must be well-defined and must not be chosen randomly. This guarantees that a simulation run with the same loaded application is reproducible after a hard reset.

——— **Caution** ———

Calling `CADIExecReset()` for any reset level must not touch any set breakpoint or unregister any registered callback object.

A call to `CADIExecReset(0)` must trigger the following behavior of the target:

• setting all registers and state variables to their initial values

• clearing all memories of the target and bring them into their initial state.

• clearing the internal list of loaded applications (because the memory is cleared). See *Application loading* on page 3-27.

After calling `CADIExecReset(0)`, it is the responsibility of the calling debugger to reload applications if that is required.

To determine the supported resets for a target, call the `CADIExecGetResetLevels()` method which provides a list with the corresponding identifiers. The contained reset level number must be forwarded to `CADIExecReset()` to trigger the required reset.

Using CADIExecReset()

`CADIExecReset()` is an asynchronous call and can therefore return before the actual reset of the target has finished. After the target has ended all required actions, the simulation thread sends out a `modeChange(CADI_EXECMODE_ResetDone)` callback to all registered debuggers. Because a target can only accept one CADI reset at a time, the calling debugger can depend on the receiving the end notification for its `CADIExecReset()` call and then proceed with other required functionality such as loading applications to the target.

——— **Note** ———

The `modeChange(CADI_EXECMODE_ResetDone)` callback is identical to the legacy `CADICallbackObj::reset()` callback.

Targets must support both callbacks to maintain backwards compatibility.

ARM recommends using `modeChange(CADI_EXECMODE_ResetDone)` in client code because the `reset()` callback is deprecated in a future version of CADI. See *CADICallbackObj::modeChange()* on page A-18.

Callback behavior

The `CADICallbackObj` class is an important part of the mechanism for controlling target execution. Unlike the interface calls of the `CADI` class that initiates behavior changes in the target, the callback mechanism reports changes in the target state back to the caller.

Some callback calls are optional and are not required for the execution control. These include:

*   semihosting (see *Using the semihosting API* on page 3-32)

*   methods provided for convenience that are not used for control, but instead enable notifying the caller to perform actions on the GUI side such as refreshing views.

Callbacks in CADI are asynchronous and can be received even if a debugger has not triggered any behavior. This is required to enable connection of multiple debuggers to a single target. If for example one debugger requests a running target to stop, all connected debuggers receive a `modeChange(CADI_EXECMODE_Stop)` callback that instructs the debuggers to change their state and to update the target views.

——— **Note** ———

Callbacks of class `CADICallbackObj` must only be called from the simulation thread. The associated debugger thread must not, either directly or indirectly, call a callback of this class.

The most important, and almost mandatory, callback for execution control is the `modeChange()` method. It reports any change of the state of the target state or if a breakpoint is hit. `modeChange()` receives the execution mode and, if required, the breakpoint ID. The typical execution modes are `CADI_EXECMODE_Run`, `CADI_EXECMODE_Stop`, `CADI_EXECMODE_Bpt`, `CADI_EXECMODE_ResetDone`, and `CADI_EXECMODE_Error`.

Issuing a `modeChange()` callback is only permitted if the state changed and the new state has been reached. For example, a change to `CADI_EXECMODE_Stop` can only be issued if the target was previously in another state, typically `CADI_EXECMODE_Run`, and the target is now in the stopped state and has finished ALL implied updates of target resources.

A change to `CADI_EXECMODE_Bpt` requires an additional breakpoint ID to inform the caller that the breakpoint has been hit. In all other cases, this parameter has to be set to zero which indicates an invalid breakpoint ID.

A mode change to `CADI_EXECMODE_Bpt` must be issued for every hit breakpoint. If multiple breakpoints triggered at the time, each of them must be reported by dedicated calls. This might be the case if, for example, a register breakpoint and a program breakpoint are hit simultaneously. Both must be reported to enable the caller to react properly to the two events.

A mode change to `CADI_EXECMODE_ResetDone` indicates the end of a CADI reset and the debugger must update all its views. The debugger might also take additional actions if the debugger was responsible for the reset. See *Controlling the execution mode* on page 3-22. The caller might expect characteristic sequences of `modeChange()` callbacks in response to a specific requested functionality. Table 3-2 on page 3-26 lists typical schemes:

Table 3-2 Typical modeChange() callback responses

Target state	Called interface method	Expected modeChange() sequence
Stopped	Debugger calls CADIExecContinue().	modeChange(CADI_EXECMODE_Run, 0)
Running	Debugger calls CADIExecStop().	modeChange(CADI_EXECMODE_Stop, 0)
Stopped	Debugger calls CADIExecSingleStep().	modeChange(CADI_EXECMODE_Run, 0) modeChange(CADI_EXECMODE_Stop, 0)
Running	Debugger calls CADIExecContinue() or CADIExecSingleStep().	No modeChange() is issued and the corresponding call returns with CADI_STATUS_TargetBusy.
Stopped	Debugger calls CADIExecStop().	No modeChange() is issued. The call returns with CADI_STATUS_OK because nothing unexpected or incorrect occurred.
Stopped	Debugger has set a program breakpoint (ID=1) to be hit. Debugger calls CADIExecContinue().	modeChange(CADI_EXECMODE_Run, 0) modeChange(CADI_EXECMODE_Bpt, 1) modeChange(CADI_EXECMODE_Stop, 0)
Stopped	Debugger has set a program breakpoint (ID=1) on the next instruction and a memory breakpoint (ID =2) on an address is modified after finishing the current instruction. Debugger calls CADIExecSingleStep() for an instruction step.	modeChange(CADI_EXECMODE_Run, 0) modeChange(CADI_EXECMODE_Bpt, 1) modeChange(CADI_EXECMODE_Bpt, 2) modeChange(CADI_EXECMODE_Stop, 0)
Stopped	Debugger has set a breakpoint (ID=1) with property continueExecution set to true. The breakpoint is hit if execution resumes. Debugger calls CADIExecContinue().	modeChange(CADI_EXECMODE_Run, 0) modeChange(CADI_EXECMODE_Bpt, 1) Target continues.
Stopped	Debugger calls CADIExecReset().	modeChange(CADI_EXECMODE_ResetDone, 0)
Running	Debugger calls CADIExecReset().	modeChange(CADI_EXECMODE_Stop, 0) if it is required that the model stop before reset. modeChange(CADI_EXECMODE_ResetDone, 0)

3.7 Application loading

Loading an application from a debugger to a target is simplified by the CADI interface. A debugger typically writes the application program code directly to the platform memory. To simplify this, CADI has a `CADIExecLoadApplication()` method that autonomously writes the application code to the target. The debugger must extract debug information, if it was provided, from the executable. This can be used to initialize more hardware resources of the simulation model such as, for example, setting the program counter to the entry point of the application.

——— **Note** ———

The file path to the binary must be visible to both the debugger and the target because only the path string is passed through the interface.

It is implementation dependent which types of executable are supported by the model. For example, some models might, or might not, support ELF files.

It is possible to load multiple applications to a target. This might, for example, be required to load several different applications to a multiprocessor. The information about each loaded application and its received command line parameters are stored in an internal list in the target.

This list always represents the currently loaded applications. To determine which applications are loaded on a connected target, call the `CADIExecGetLoadedApplications()`. It returns all information, including the file paths and the applied command line parameters, used to load the corresponding binary. This data can be used by other debuggers connecting to this processor to obtain the required debug information.

The list of loaded applications must be preserved until a hard reset, that is until `CADIExecReset(level=0)`. Other reset levels that modify program memory can also empty this list. See the documentation for the model to determine the model behavior.

——— **Note** ———

A simple `CADIMemWrite()` does not have an impact on the list of loaded applications even if it breaks one of them.

To unload an application from the target (or even better, to invalidate the application) without using a CADI reset, the debugger can call `CADIExecUnloadApplication()`. This removes the application information and, if loaded, the debug information from the target. Memory contents are not, however, erased by this call. The passed file path must be identical with the one used for `CADIExecLoadApplication()`.

——— **Note** ———

It is implementation dependent if the model supports handling debug information. This is typically not required as the debugger side must extract the information from the application image anyway.

3.8   CADI Disassembler

The CADI Disassembler is an extension of the common CADI interface. It enables a debugger to exploit a disassembler that is integrated into a simulation model. This has the advantage of entirely separating the ISA-specific information from the implementation of the debugger.

A CADI Disassembler is mainly intended to deliver disassembly information from a target to the debugger. However It also provides interface methods that expose debug information a model might have extracted.

The CADI Disassembler interface consists of the `CADIDisassembler` class and the `CADIDisassemblerCB` class. `CADIDisassemblerCB` is required to be implemented by the connected debugger and declares callback methods. These are directly linked to methods in `CADIDisassembler` and return the requested information to those calls.

3.8.1   Obtaining a CADI Disassembler

A pointer to a certain `CADIDisassembler` object is obtained from the corresponding `CAInterface` instance in the target. See *CADI classes used to control the simulation target* on page 1-7.

—— **Note** ——

The `CADIGetDisassembler()` method of the corresponding `CADI` object is retained only for compatibility with former CADI versions. It is deprecated and must not be used in new implementations.

3.8.2   CADI Disassembler callbacks

The CADI Disassembler interface provides a callback mechanism that requires an appropriate implementation in the debugger. The callback mechanism, unlike other callback mechanisms in CADI, is not intended to enable an asynchronous behavior. The CADI Disassembler calls that trigger callbacks are intended to be synchronous and all issued callbacks must be finished by the time the calling method returns.

The CADI Disassembler callbacks provide a way to return the requested disassembly information in character strings of arbitrary size without passing ownership of the corresponding data across library boundaries. Using this mechanism, the debugger receives a string buffer owned by the target and creates a local copy.

—— **Note** ——

The string must be null terminated because the length of the issued string is not explicitly passed to the debugger.

The relationships between the methods of `CADIDisassembler` and the related callback methods are listed in Table 3-3:

Table 3-3 Relationships between CADIDisassembler and the callback methods

CADIDisassembler	CADIDisassemblerCB
`GetModeNames()`	`ReceiveModeName()`
`GetDisassembly()`	`ReceiveDisassembly()`
`GetSourceReferenceForAddress()`	`ReceiveSourceReference()`

In contrast to other callback mechanisms, the pointer to the utilized callback object is not registered to the disassembler instance but explicitly passed to it with each call.

3.8.3 Disassembly Modes

The CADI Disassembler interface supports different disassembly modes. Such modes might, for example, represent different instruction sets that are supported by a processing unit. A simple example is an ARM processor that supports the A32 instruction set and the T32 instruction set.

A debugger can use the `GetModeCount()` and the `GetModeNames()` methods to determine which modes are supported. Typically all CADI Disassembler implementations support at least one mode which can be considered as a *don't care* mode. The ID for this mode is reserved as 0. The mode ID enables the instruction at the requested address to be disassembled with consideration of the instruction set and the current mode for the processing unit.

——— **Caution** ———

Querying the disassembly for a specific memory address with a nonzero mode ID results in the interpretation of the memory contents according to the instruction set for that mode. The disassembler proceeds even if it is an instruction of a different set. This might lead to an incorrect, but apparently successful, disassembly if the memory contents accidentally represents a valid instruction in the ISA for the other mode.

3.8.4 CADIDisassemblerStatus

Similar to the `CADI` class, the `CADIDisassembler` class can indicate the success or failure of some methods with the dedicated status type `CADIDisassemblerStatus`. This enumeration type informs the debugger about more details of an uncompleted method call. Methods that use this return type are those that request disassembly information. The following return values are defined:

`CADI_DISASSEMBLER_STATUS_OK`

>   The method call succeeded. All requested information was sent to the debugger either by callbacks or by filling a provided data buffer.

>   For multiply-triggered callbacks, for example when requesting multiple subsequent instructions to be disassembled, all have been issued to the debugger before returning from the method call.

`CADI_DISASSEMBLER_STATUS_NO_INSTRUCTION`

>   Disassembling the requested address failed because the data was not a valid instruction for the specified ISA.

`CADI_DISASSEMBLER_STATUS_ILLEGAL_ADDRESS`

>   Disassembling the requested address failed because it was not within a valid memory range of the target.

>   Reading memory from this address with `CADI::CADIMemRead()` also fails.

`CADI_DISASSEMBLER_STATUS_ERROR`

>   An error occurred that is not covered by one of the other return values. This might be, for example, because of a lost connection or an illegal method call parameter such as, for example, an invalid pointer to a callback object.

3.8.5 Acquiring Disassembly

Call `GetDisassembly()` to get the disassembly from a CADI Disassembler, The method has the following parameters:

callback   The callback object for the debugger to use to return the disassembly information.

address   The address the disassembly starts from.

nextAddr   Used by the disassembler to return the next address that can be disassembled. This gives the debugger a hint where to continue with disassembling after the last instruction of the current request.

           This information is particularly useful for uncompleted calls. It gives the debugger an address from which it can resume.

mode   The mode used to disassemble the data. This can either be an explicitly selected mode or the mode the processing unit is currently in. For the latter case, the *don't care* ID of 0 must be forwarded.

desiredCount The number of instructions for the disassembler to process. This must also be the maximum number of `ReceivedDisassembly()` callbacks issued.

——— **Note** ———

The `desiredCount` refers to the number of requested instructions. If the mode ID is 0, the size of the instruction words can vary if the mode changes in between. It is therefore possible that the distance between the addresses (as returned by the callback) is not equally spaced.

Updating nextAddr after the last instruction is reached

If the last valid instruction within a memory space is reached, `nextAddr` must be set to this last instruction. The last valid instruction can be determined by testing the following conditions:

* `nextAddr` is identical to the requested address

* the `GetDisassembly()` call returns with `CADI_DISASSEMBLER_STATUS_OK` and triggers only one `ReceiveDisassembly()` callback no matter how many instructions are requested.

Figure 3-4 on page 3-31 shows a call where the last instruction in the range is a valid instruction.

Target			Debugger
Program memory		1. Call GetDisassembly() with address as 0x0 and desiredCount as 0x200	

0x0	Valid instruction
	. . .
0xFF	Valid instruction
0x100	No instruction
	. . .
0x1FF	No instruction
0x200	Valid instruction
	. . .

2. ReceiveDisassembly() called 256 (0x100) times

3. nextAddr of GetDisassembly() is set to 0x200

4. GetDisassembly() returns with status as CADI_DISASSEMBLER_STATUS_OK

5. Call GetDisassembly() with the address set to nextAddr from the previous call.

6. GetDisassembly() returns a value for nextAddr that is different from address.

Figure 3-4 nextAddr set to last instruction

Figure 3-5 shows a call where the last instruction in the range is the last instruction in the memory space.

Target			Debugger
Program memory		1. Call GetDisassembly() with address as 0x100 and desiredCount as 0x200	

0x0	No instruction
	. . .
0xFF	No instruction
0x100	Valid instruction
	. . .
0x1FF	Valid instruction
0x200	No instruction
	. . .

2. ReceiveDisassembly() called 256 (0x100) times

3. nextAddr of GetDisassembly() is set to 0x1FF

4. GetDisassembly() returns with status as CADI_DISASSEMBLER_STATUS_OK

5. Call GetDisassembly() with address set to the same value of nextAddr from the previous call.

6. GetDisassembly() returns a value for nextAddr equal to address. Only one ReceiveDisassembly() call is issued no matter how many instructions were requested.

Figure 3-5 nextAddr set to last valid instruction

3.9 Using the semihosting API

CADI provides a semihosting interface that enables interaction between a user and a connected target. As shown in Figure 3-6, a debugger can use the host machine I/O to emulate the I/O devices in a simulation platform. An application running on a target component can request keyboard input that is then provided interactively when you enter the input on the host keyboard.
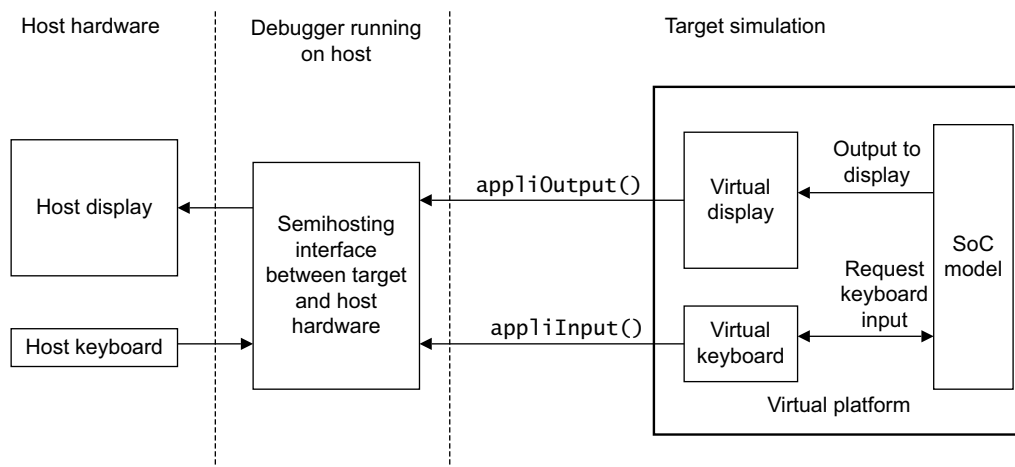
Figure 3-6 Semihosting interface

Because semihosting is used by the simulation target to provide and receive information, the interface methods are provided by the `CADICallbackObj` object. The primary methods are `appliInput()` and `appliOutput()`. See *CADICallbackObj::appliInput() on page A-17* and *CADICallbackObj::appliOutput() on page A-17*. Both use a data buffer of type `char` and the buffer size defined by the target.

After the call returns, the `actualCount` parameter indicates:

*   how many characters were successfully written to the output device by `appliOutput()`

*   how many characters were received from the input device by `appliInput()`.

Because the forwarded string might contain `'\0'` characters, the end of the string is not indicated by `'\0'`.

———— **Note** ————

`actualCount` is also used to indicate:
*   that the end of file was reached by returning zero
*   that a string reading error occurred by returning `static_cast<uint32_t>(-1)`.

The addressed target of `appliInput()` and `appliOutput()` is typically one `StdIn`, `StdOut`, and `StdErr` streams on the host. The host can redirect these standard stream calls to log files. The IDs for the standard streams are defined in the enumeration type `CADIStreamId`. The numbering corresponds to the C file conventions:

*   0 is `stdin`
*   1 is `stdout`
*   2 is `stderr`
*   IDs greater than 2 identify explicitly opened file streams.

Use the `appliOpen()` and `appliClose()` callbacks to open and close streams to files. The returned ID identifies the stream. The file stream IDs and standard stream IDs cannot overlap.

The semihosting interface also provides the `doString()` method to send messages from the target to the caller. This method can be used, for example, to send error messages or debug output. This call is not intended to be used for passing printouts from an application. See *CADICallbackObj::doString()* on page A-18.

3.10 Profiling

These APIs give access to execution and memory debug-profiling for a processor:

- *CADIProfilingCallbacks::profileResourceAccess()* on page A-52
- *CADIProfilingCallbacks::profileRegisterHazard()* on page A-52
- *CADIProfiling::CADIProfileSetup()* on page A-54
- *CADIProfiling::CADIProfileControl()* on page A-55
- *CADIProfiling::CADIProfileTraceControl()* on page A-55
- *CADIProfiling::CADIProfileTraceControl()* on page A-55
- *CADIProfiling::CADIProfileGetExecution()* on page A-55
- *CADIProfiling::CADIProfileGetMemory()* on page A-56
- *CADIProfiling::CADIProfileGetTrace()* on page A-56
- *CADIProfiling::CADIProfileGetRegAccesses()* on page A-57
- *CADIProfiling::CADIProfileSetRegAccesses()* on page A-57
- *CADIProfiling::CADIProfileGetMemAccesses()* on page A-58
- *CADIProfiling::CADIProfileSetMemAccesses()* on page A-58
- *CADIProfiling::CADIProfileGetAddrExecutionFrequency()* on page A-59
- *CADIProfiling::CADIProfileSetAddrExecutionFrequency()* on page A-59
- *CADIProfiling::CADIGetNumberOfInstructions()* on page A-60
- *CADIProfiling::CADIProfileInitInstructionResultArray()* on page A-60
- *CADIProfiling::CADIProfileGetInstructionExecutionFrequency()* on page A-60
- *CADIProfiling::CADIProfileGetInstructionExecutionFrequency()* on page A-60
- *CADIProfiling::CADIProfileGetInstructionExecutionFrequency()* on page A-60
- *CADIProfiling::CADIProfileSetInstructionExecutionFrequency()* on page A-61
- *CADIProfiling::CADIRegisterProfileResourceAccess()* on page A-61
- *CADIProfiling::CADIUnregisterProfileResourceAccess()* on page A-61
- *CADIProfiling::CADIProfileRegisterCallBack()* on page A-61
- *CADIProfiling::CADIProfileUnregisterCallBack()* on page A-61.

————— **Note** —————

The `CADIProfiling` class is not implemented by the processor models in the Fast Models Portfolio. It is not, therefore, covered in detail in this guide.

This API is for debug profiling such as, for example, tracing program execution. It is not related to the *ESL Cycle Accurate Profiling Interface* (CAPI).

Chapter 4
The CADI Extension Mechanism

This chapter describes the CADI extension mechanism used to add interfaces to a target and the modifications that are required on both caller side and target side. It contains the following sections:

4.1   Overview of the extension mechanism

One of the major features introduced with CADI 2.0 is the extension mechanism that:

*   provides a simple framework that enables adding more interfaces to a target component

*   enables checking compatibility between the caller and the target.

A single target can present multiple interfaces. Each of the interfaces, including the basic `CADI` interface, is an extension of the abstract `CAInterface` class. The client can use a pointer to any of the interfaces to obtain a pointer to any of the other interfaces implemented by the target.

The CADI extension mechanism is based on the `CAInterface` class and its methods that must be implemented for any custom interface:

`IFNAME()`   is a static method that must be defined by each interface class. It returns the name.

`IFREVISION()` is a static method that must be defined by each interface class. It returns the revision.

`ObtainInterface()`

is a virtual method that is implemented in the class that implements the interface. It retrieves an interface from a target, including those introduced by an extension, and performs compatibility checks.

The main work of adding a custom extension to CADI must be done in the implementation for the target. A new class is declared and implemented provides access to all interfaces the target component offers.

A typical implementation requires consideration of the following:

1.   Declare a class with the custom interface extensions that must be derived from `CAInterface`. The inherited method calls must be implemented.

2.   Implement `ObtainInterface()` for the custom extension so that all existing interfaces are accessible.

3.   Other implemented interfaces provided by the target require a link to the extension through their `ObtainInterface()` implementations.

Implementation example for extending the target-side on page 4-3 provides an example of creating and implementing an extension interface.

4.2 Implementation example for extending the target-side

This section describes:

- one way to create a simple extension interface
- the required steps to use the extension mechanism in an implementation.

The class relationships are shown in Figure 4-1:
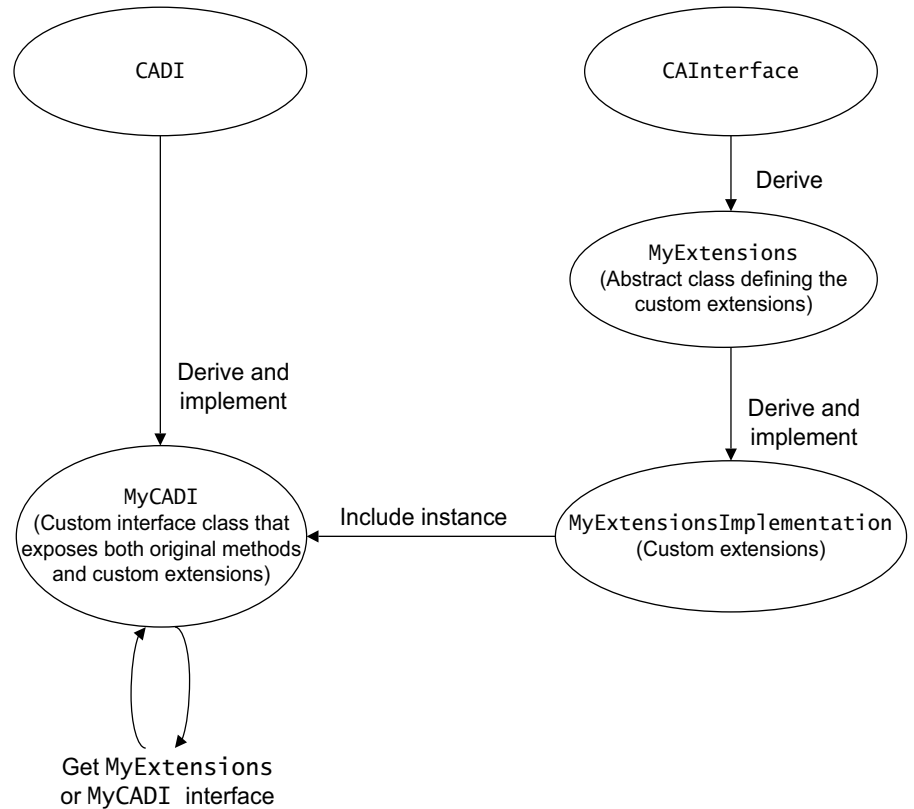
Figure 4-1 Custom extensions to a CADI interface class

To create the target-side implementation:

1. Declare the interface that provides the custom extensions in a new class. The class is called `MyExtensions` in Example 4-1:

Example 4-1 MyExtensionsAbstract class

```
// MyExtensions Interface Class
// Keep this class as abstract as possible. It should be
// the interface declaration, only.

class MyExtensions
    : public eslapi::CAInterface
{
  public:
      static eslapi::if_name_t IFNAME()
          { return "MyExtensions"; }
      static eslapi::if_rev_t IFREVISION()
          { return 0; }

      virtual eslapi::CAInterface*
```

```
        ObtainInterface(eslapi::if_name_t ifName,
                        eslapi::if_rev_t  minRev,
                        eslapi::if_rev_t* actualRev);
  public:
      virtual void MyMethod1() = 0;
      virtual void MyMethod2() = 0;
      ...
}
```

The `MyExtensions` class is derived from `eslapi::CAInterface` to enable the extension mechanism. It must implement the `IFNAME()` and `IFREVISION()` methods. The remainder of the interface must be kept as abstract as possible provide a clean separation between the interface declaration and the interface implementation.

2. It might not be obvious, but a CADI target that receives a custom extension interface also provides an implementation of the CADI interface itself. Enabling access to the custom extensions requires modification of the CADI implementation and, for Example 4-2, the passing of pointers from the instantiated interface objects.

Example 4-2 shows the declaration of the class `MyExtensionsImplementation` that provides the actual implementation of the custom interface and some additions required to *mount* the new interface. The class `MyCADI` is the class derived from `CADI` as shown in Example 4-1 on page 4-3.

Example 4-2 Declaration of MyExtensionsImplementation

```
// MyExtensionsImplementation Class
// Implementing MyExtensions interface.
class MyExtensionsImplementation
    : public MyExtensions
{
private:
    MyCADI *myCadiPointer; /* Pointer to an object of MyCADI which is
                              required for the link to the original interfaces.*/
public:
    //Called by MyCADI constructor.
    MyExtensionsImplementation(MyCADI *myCadi)
    { myCadiPointer = myCadi; }

    static eslapi::if_name_t IFNAME()
    { return "MyExtensionsImplementation"; }

    static eslapi::if_rev_t IFREVISION()
    { return 0; }

    virtual eslapi::CAInterface*
    ObtainInterface(eslapi::if_name_t ifName,
                    eslapi::if_rev_t minRev,
                    eslapi::if_rev_t* actualRev);

public:
    void MyMethod1();
    void MyMethod2();
};
```

This class owns a pointer of class `MyCADI` that points to the instance of the CADI implementation linked to this custom extension. This is required by the `ObtainInterface()` method as shown in Example 4-3 on page 4-5. In this example, the corresponding pointer is passed through the constructor.

This class must implement its own versions of the `IFNAME()` and `IFREVISION()` methods

3. After declaring the MyExtensionsImplementation class, implement the inherited `ObtainInterface()` that receives the following parameters:

   ifname   The interface name requested by the caller such as, for example, `MyExtensionsImplementation`.

   minRev   The minimum revision required by the caller. Use 0 to accept any revision.

   actualRev The actually implemented revision (greater than or equal to `minRev`). This value must be set by the target.

   Example 4-3 shows a typical implementation of `ObtainInterface()` for the extension:

Example 4-3 ObtainInterface()

```
// Call ObtainInterface of MyCADI which is derived from CADI.
// This guarantees that, for example, an ObtainInterface()
// call for "eslapi.CAInterface" returns the same pointer
// from MyExtensionsImplementation AND from MyCADI.
eslapi::CAInterface*
MyExtensionsImplementation::ObtainInterface(eslapi::if_name_t ifName,
                                            eslapi::if_rev_t minRev,
                                            eslapi::if_rev_t* actualRev)
{
    return myCadiPointer->ObtainInterface(ifName, minRev, actualRev);
}
```

This implementation forwards the interface request directly to the modified CADI implementation. The reasoning is to implement `ObtainInterface()` in exactly one place so that only one implementation must be edited if custom interfaces must be added. The `CAInterface` specification requires that the same pointer is provided for the specific requested interface, for example `eslapi.CAInterface`, for any call of `ObtainInterface()` from any class such as `MyExtensionsImplementation::ObtainInterface()` or `myCadiPointer::ObtainInterface()`. Because there is only one place to return these pointers, it can be guaranteed that the pointer for a requested interface is always the same.

4. The final step to implement and mount a custom extension interface is to modify the existing CADI implementation by deriving it and the required code as shown in Example 4-4:

Example 4-4 Changes to MyCADI class

```
// MyCADI Class
// Derived from class CADI. The main purpose is to
// provide the modified ObtainInterface() method.
class MyCADI
    : public eslapi::CADI
{
private:
    MyExtensionsImplementation* myExtensionsPointer;

public:
    MyCADI()
    { myExtensionsPointer = new MyExtensionsImplementation(this); }

    static eslapi::if_name_t IFNAME()
    { return "MyCADI"; }
```

```
                              static eslapi::if_rev_t IFREVISION()
                              { return 0; }

                              virtual eslapi::CAInterface*
                              ObtainInterface(eslapi::if_name_t ifName,
                                              eslapi::if_rev_t minRev,
                                              eslapi::if_rev_t* actualRev);

                    // …
                };
```

In Example 4-4 on page 4-5, an instance of `MyExtensionsImplementation` is owned by `MyCADI`. This is instantiated in the class constructor and accessed through a pointer. It is required to support calling the `ObtainInterface()` implementation to return one of the interfaces such as `MyExtensionsImplementation` or `MyExtensions`.

The value of the `MyExtensionsImplement` class listed in Example 4-4 on page 4-5 is the implementation of `ObtainInterface()` in the code fragment listed in Example 4-5:

Example 4-5 Using ObtainInterface()

```cpp
// MyCADI has been chosen to provide the "central" ObtainInterface()
// method, i.e. all ObtainInterface() calls arriving in the target
// are routed to this implementation. This requires a corresponding
// check for all interfaces and pointers to all available interface
// instances. In this example we have to check for:
// - MyCADI
// - MyExtensionsImplementation
// - MyExtensions
// - CADI
// - CAInterface
eslapi::CAInterface*
MyCADI::ObtainInterface(eslapi::if_name_t ifName,
                        eslapi::if_rev_t minRev,
                        eslapi::if_rev_t* actualRev)
{
    // Check if queried interface is "MyCADI" and if the
    // provided revision is sufficient.
    if((strcmp(ifName,IFNAME()) == 0)
       && (minRev <= IFREVISION()))
    {
        if (actualRev != NULL) // NULL pointer check
        {
            *actualRev = IFREVISION(); // Set the actual rev
        }
        return this;
    }

    // Check if queried interface is "MyExtensionsImplementation" and
    // if the provided revision is sufficient.
    if((strcmp(ifName, MyExtensionsImplementation::IFNAME()) == 0)
       && (minRev <= MyExtensionsImplementation::IFREVISION()))
    {
        if (actualRev != NULL) // NULL pointer check
        {
            *actualRev = MyExtensionsImplementation::IFREVISION();// Set the actual rev
        }
  // This is an additional check added for MyExtensionsImplementation,
  // return the corresponding pointer.
        return myExtensionsPointer;
```

```
    }

    // Check if queried interface is "MyExtensions" and if the
    // provided revision is sufficient.
    if((strcmp(ifName, MyExtensions::IFNAME()) == 0)
       && (minRev <= MyExtensions::IFREVISION()))
    {
        if (actualRev != NULL) // NULL pointer check
        {
    *actualRev = MyExtensions::IFREVISION(); // Set the actual rev
        }
  // This is an additional check added for MyExtensionsImplementation,
  // return the corresponding pointer.
        return myExtensionsPointer;
    }

    // Check if queried interface is "CADI" and if the
    // provided revision is sufficient.
    if((strcmp(ifName, eslapi::CADI::IFNAME()) == 0)
       && (minRev <= eslapi::CADI::IFREVISION()))
    {
        if (actualRev != NULL) // NULL pointer check
        {
    *actualRev = eslapi::CADI::IFREVISION(); // Set the actual rev
        }
        return this;
    }

    // Check if queried interface is "CAInterface" and if the
    // provided revision is sufficient.
    if((strcmp(ifName, eslapi::CAInterface::IFNAME()) == 0)
       && (minRev <= eslapi::CAInterface::IFREVISION()))
    {
        if (actualRev != NULL) // NULL pointer check
        {
*actualRev = eslapi::CAInterface::IFREVISION();// Set the actual rev
        }
        return this;
    }

    // Target does not provide the requested interface.
    return NULL;
}
```

Example 4-5 on page 4-6 shows that this ObtainInterface() implementation is very similar to the common one. This example, however, has two interface checks associated with the added myExtensionsPointer pointer.

These interface checks are similar to the usual checks, but if one of the two interfaces is recognized, ObtainInterface() does not return the this pointer, but instead returns the pointer to the instantiated extension implementation myExtensionsPointer.

4.3 Obtaining a custom interface

Implementation example for extending the target-side on page 4-3 described how to implement a custom extension for the CADI interface on target-side. This interface must now be obtained by the caller. Follow the procedure described in this section to ensure the correct functionality of the acquired interface and to avoid, for example, the utilization of an outdated interface revision.

The procedure of obtaining a custom interface is the same as the one described in *Obtaining an interface pointer to the target* on page 3-6 for the standard interfaces:

1. A CAInterface pointer to the target interface class is required. This is typically returned by the CADI simulation.

2. The ObtainInterface() method must be called to check if the required interface is provided.

3. The returned pointer to CAInterface, which might differ from the originally obtained one, must be converted to a pointer to the requested interface class by using a static_cast().

    Example 4-6 shows how this might be done for the MyExtensions and MyCADI classes as implemented in *Implementation example for extending the target-side* on page 4-3.

Example 4-6 Using CADISimulation to return a pointer to the interface

```
CADISimulation* cadiSimulation;
uint32_t targetID;
CAInterface* ca_interface;
MyExtensions* my_extensions_if;
        .
        .
        .
   //get the CADISimulation pointer
        .
        .
        .
// here, gets a pointer of type CAInterface. this pointer can be used to obtain
// any interface provided by the target using ObtainInterface()
ca_interface = cadiSimulation->GetTarget(targetID);

//obtain the desired interface
if_name_t ifName = "MyExtensions";
if_rev_t  minRev = 0;
if_rev_t  actualRev = 0;


// ObtainInterface() asks for "MyExtensions" interface
// it returns the corresponding base class pointer
ca_interface = ca_interface->ObtainInterface(ifName, minRev, &actualRev);
if (ca_interface == NULL)
{
   //something went wrong, handle it...
}
else //MyExtensions interface supported
{
   my_extensions_if =
           static_cast<MyExtensions*>(ca_interface);
}
```

```
.
.
.   //go on using the obtained interface extensions
```

Appendix A
Class Reference

This chapter describes the classes used to create, initialize, and communicate with a simulation. It contains the following sections:

───── **Note** ─────

Implementing the `CADIDisassemblerCB`, `CADIDisassembler`, `CADIProfilingCallbacks`, and `CADIProfiling` classes and the methods that use them is optional. They are typically only used for components that execute applications.

─────────────

A.1 The CAInterface class

This section describes the CAInterface class that is the base class for all CADI classes.

- *About the CAInterface class*
- *CAInterface class declaration* on page A-3
- *CAInterface::IFNAME()* on page A-3
- *CAInterface::IFNAME()* on page A-3
- *CAInterface::ObtainInterface()* on page A-3

A.1.1 About the CAInterface class

CAInterface provides a basis for a software model built around components and interfaces.

For CADI, an interface:

- is an abstract class consisting entirely of pure virtual methods

- derives from CAInterface

- provides a number of methods for interacting with a component.

- are identified by a string name of type if_name_t and an integer revision of type if_rev_t. A higher revision number indicates a newer revision of the same interface.

A component is a black-box entity that has a unique identity and provides concrete implementations of one or more interfaces:

- Each of these interfaces can expose different facets of the component behavior.

- These interfaces are the only way to interact with the component.

- There is no way for a client to enumerate the set of interfaces that a component implements. The client must ask for specific interfaces by name.

  (The implementation of a component interface might be provided by one or several interacting C++ objects. This is an implementation detail that is transparent to the client.)

- If the component does not implement the requested interface, it returns a NULL pointer.

The CAInterface class is the base class for all interfaces. It defines a method, CAInterface::ObtainInterface(), that enables a client to obtain a reference to any of the interfaces that the component implements.

The client specifies the ID and revision of the interface that it is requesting. The component can return NULL if it does not implement that interface, or only implements a lower revision.

Because each interface derives from CAInterface, a client can call ObtainInterface() on any one interface pointer to obtain a pointer to any other interface implemented by the same component.

The following rules govern the use of components and interfaces:

- Each component is distinct. No two components can return the same pointer for a given interface. An ObtainInterface() call on one component must not return an interface on a different component.

- Each interface consists of a name, a revision number, and a C++ abstract class definition. The return value of ObtainInterface() is either NULL or a pointer, castable to the class type.

- Where two interfaces have the same if_name_t, the newer revision of the interface must be compatible with the old revision. (This includes the binary layout of any data-structures that it uses and the semantics of any methods.)

- During the lifetime of a component, any calls to ObtainInterface() for a given interface name and revision must always return the same pointer value. It must not matter which of the component interfaces is used to invoke ObtainInterface().

- All components must implement an interface derived from eslapi::CAInterface.

A.1.2   CAInterface class declaration

The CAInterface class is the base class for all CADI interface classes.

Example A-1 The CAInterface class

```
class ESLAPI_WEXP CAInterface
   {
   public:
       static if_name_t IFNAME() { return "eslapi.CAInterface"; }
       static if_rev_t IFREVISION() { return 0; }
       virtual ~CAInterface() {}
   public:
       virtual CAInterface * ObtainInterface(if_name_t ifName,
               if_rev_t minRev, if_rev_t * actualRev) =0;
   };
```

A.1.3   CAInterface::IFNAME()

The default declaration for IFNAME() is:

```
static if_name_t IFNAME() { return "eslapi.CAInterface"; }
```

The component interface overrides this method to provide the name for the specific interface.

A.1.4   CAInterface::IFREVISION()

The default declaration for IFREVISION() is:

```
static if_rev_t IFREVISION() { return 0; }
```

The component interface overrides this method to provide the revision number for the specific interface.

A.1.5   CAInterface::ObtainInterface()

ObtainInterface() enables a client to obtain a reference to any of the interfaces that the component implements. The default declaration is:

```
virtual CAInterface * ObtainInterface(if_name_t ifName,
                                      if_rev_t minRev,
                                      if_rev_t * actualRev) =0;
```

where:

if_name_t	is a name identifying the requested interface
minRev	specifies the minimum minor revision required
actualRev	if not NULL, on return holds the actual revision number implemented
return value	is a pointer to the requested interface, or NULL.

A.2 The CADIBroker class

This section describes the CADIBroker class that enables connecting to existing simulations and creating new simulations:

- *CADIBroker class definition*
- *Creating the CADIBroker* on page A-5
- *CADIBroker::GetSimulationFactories()* on page A-5
- *CADIBroker::GetSimulationInfos()* on page A-6
- *CADIBroker::SelectSimulation()* on page A-6
- *CADIBroker::Release()* on page A-7.

A.2.1 CADIBroker class definition

The CADIBroker class definition is shown in Example A-2.

——— **Note** ———

The CADI broker owns all CADI simulations and no other class is permitted to delete them.

If a CADI factory creates a simulation, it must transfer the pointer to the new simulation to the broker.

If the simulation is shut down or killed, the broker is responsible for deleting the simulation. This must be done by processing GetSimulationInfos() and checking for running simulations (check that the reference count is 0 and any other implementation-specific conditions are in the appropriate state).

Example A-2 The CADIBroker class

```
class WEXP CADIBroker: public CAInterface
{
public:
static if_name_t IFNAME() { return "eslapi.CADIBroker2"; }
static if_rev_t IFREVISION() { return 0; }

virtual ~CADIBroker() {}

virtual void Release() = 0;

virtual CADIReturn_t GetSimulationFactories(uint32_t startFactoryIndex,
    uint32_t desiredNumberOfFactories, CADISimulationFactory **factoryList,
    uint32_t *actualNumberOfFactories) = 0;

virtual CADIReturn_t GetSimulationInfos(uint32_t startSimulationInfoIndex,
    uint32_t desiredNumberOfSimulations, CADISimulationInfo_t *simulationList,
    uint32_t *actualNumberOfSimulations) = 0;

virtual CADISimulation* SelectSimulation( uint32_t simulationId,
    CADIErrorCallback* errorCallbackObject, CADISimulationCallback*
    simulationCallbackObject,
    char simulationCallbacksEnable[CADI_SIM_CB_Count])=0;
};
```

A.2.2   Creating the CADIBroker

Example A-3 shows the prototypes for the functions that create the `CADIBroker`. This is the first step in creating a new simulation or connecting to an existing one.

Example A-3 Creating the CADIBroker

```
extern "C"
{
// Global function exported by a dynamically loaded object.
// This function must exist in a dynamically loaded object(DLL/.so).
// It allows the client to instantiate the CADIBroker.
CADI_WEXP eslapi::CADIBroker * CreateCADIBroker();
}
```

The prototype declaration listed in Example A-4 enables a global function to instantiate a broker from a dynamically loaded object.

Example A-4 CADIBroker type declaration

```
typedef CADIBroker * (CreateCADIBroker_t)();
```

Clients must locate this symbol and cast it as a pointer to `CreateCADIBroker_t`:

```
// CreateCADIBroker_t
void * entry = lookup_symbol(dll, "CreateCADIBroker");
CADIBroker *broker = ((*CADIBroker::CreateCADIBroker_t)entry)();
```

A.2.3   CADIBroker::GetSimulationFactories()

Returns a list of possible simulation factories provided by this simulation broker. This list is static for a given `CADIBroker`.

```
virtual CADIReturn_t CADIBroker::GetSimulationFactories(
                     uint32_t startFactoryIndex,
                     uint32_t desiredNumberOfFactories,
                     CADISimulationFactory **factoryList,
                     uint32_t *actualNumberOfFactories) =0;
```

where:

`startFactoryIndex`

> is the index of first factory to return from the internal list maintained by the broker. If `startFactoryIndex` exceeds the maximum factory index, `CADI_STATUS_IllegalArgument` is returned.

`desiredNumberOfFactories`

> is the required number of factories to return.

>> ——— **Caution** ———
>> The `factoryList` array must be at least this size.

`factoryList`   is the array of factory pointers returned by this call. This array must be allocated by caller with a minimum size of `desiredNumberOfFactories`.

— **Note** —

The returned factory pointers must not be used to delete the factories. The factories are owned by the broker.

`actualNumberOfFactories`

is the actual number of factories returned.

A.2.4 CADIBroker::GetSimulationInfos()

Returns a list of simulation infos informing about the running simulations managed by this CADI simulation broker.

— **Note** —

This list can change dynamically during lifetime of this `CADIBroker`.

```
virtual CADIReturn_t CADIBroker::GetSimulationInfos(
                  uint32_t startSimulationInfoIndex,
                  uint32_t desiredNumberOfSimulations,
                  CADISimulationInfo_t *simulationList,
                  uint32_t *actualNumberOfSimulations) =0;
```

where:

`startSimulationInfoIndex`

is the index of the first simulation info, within the internal list of running simulators, to return.

If `startSimulationInfoIndex` exceeds the maximum simulation info index, `CADI_STATUS_IllegalArgument` is returned.

`desiredNumberOfSimulations`

is the required number of simulation infos to return.

— **Caution** —

Array `simulationInfoList` must have at least this size.

`simulationList`

is the array of simulation infos returned by this call. This array must be allocated by the caller.

— **Note** —

The minimum size of this array is `desiredNumberOfSimulationInfos`.

`actualNumberOfSimulations`

is the actual number of simulation infos returned.

A.2.5 CADIBroker::SelectSimulation()

This method enables connecting to the running simulation selected by the simulation identifier. A pointer to the simulation is returned on success. If no simulation with the given ID is managed by this broker, 0 is returned.

```
virtual CADISimulation* CADIBroker::SelectSimulation( uint32_t simulationId,
                          CADIErrorCallback* errorCallbackObject,
                          CADISimulationCallback* simulationCallbackObject,
                          char simulationCallbacksEnable[CADI_SIM_CB_Count]) =0;
```

where:

simulationId is the ID of the simulation to be returned. This is part of the respective entry in
the list of the simulation infos `simulationList` returned by `GetSimulationInfos()`.

errorCallbackObject

is the error callback object to be used for signaling error conditions.

simulationCallbackObject

is the simulation callback object to be used for signaling model-wide conditions.

——— **Note** ———

This callback might be called during execution of `SelectSimulation()` to, for
example, signal that the simulation wants to shut down.

simulationCallbacksEnable

The elements of this array enable or disable specific simulation callbacks. The
simulation must always check if the callbacks are enabled and these must not be
called if they are disabled. The callbacks might be disabled, for example, if the
listener does not want to be called in certain cases.

return value

is the pointer to the simulation or `NULL` if the call fails.

A.2.6   CADIBroker::Release()

Release this broker. A debugger is expected to release the `CADIBroker` at the end of a debugging
session. The debugger must manage releasing all obtained `CADIFactories` before finally
destroying the broker. An obtained `CADI` interface of a running simulation must be released
before destroying the broker.

```
virtual void Release() =0;
```

A.3 The CADISimulationFactory class

This section describes the `CADISimulationFactory` class that provides a mechanism to start new simulations:

A.3.1 CADISimulationFactory class definition

The `CADISimulationFactory` definition is listed in Example A-5.

Example A-5 CADISimulationFactory class

```
class CADI_WEXP CADISimulationFactory : public CAInterface
{
public:
    // Return the CAInterface name for this interface.
    static if_name_t IFNAME() { return "eslapi.CADISimulationFactory2"; }
    // Specify the current minor revision for this interface.
    static if_rev_t IFREVISION() { return 0; }
    virtual void Release() = 0;
    virtual const char* GetName() = 0;
    virtual const char* GetDescription() = 0;
    virtual CADIReturn_t GetParameterInfos(uint32_t startParameterInfoIndex,
                    uint32_t desiredNumberOfParameterInfos,
                    CADIParameterInfo_t *parameterInfoList,
                    uint32_t *actualNumberOfParameterInfos) = 0;
    virtual CADISimulation *Instantiate(CADIParameterValue_t *parameterValues,
                     CADIErrorCallback *errorCallbackObject,
                     CADISimulationCallback *simulationCallbackObject,
                     char simulationCallbacksEnable[CADI_SIM_CB_Count]) = 0;
};
```

A.3.2 CADISimulationFactory::Release()

Release this simulation factory. A debugger is expected to release the simulation factory as soon as the CADI target is obtained.

```
virtual void CADISimulationFactory::Release() =0;
```

A.3.3 CADISimulationFactory::GetName()

Returns the name for this factory.

```
virtual const char * CADISimulationFactory::GetName() =0;
```

A.3.4 CADISimulationFactory::GetDescription()

Returns the description for this factory.

```
virtual const char * CADISimulationFactory::GetDescription() =0;
```

A.3.5   CADISimulationFactory::GetParameterInfos()

Returns a list of simulation parameters and their attributes that must be set through corresponding values in the `Instantiate()` call of this class.

```
virtual CADIReturn_t CADIBroker::GetParameterInfos(
                    uint32_t startParameterInfoIndex,
                    uint32_t desiredNumberOfParameterInfos,
                    CADIParameterInfo_t *parameterInfoList,
                    uint32_t *actualNumberOfParameterInfos) =0;
```

where:

`startParameterInfoIndex`

> is the index of the first parameter info to return. If `startParameterInfoIndex` exceeds the maximum simulation info index, `CADI_STATUS_IllegalArgument` is returned

`desiredNumberOfParameterInfos`

> is the required number of parameter infos to return.

> ——— **Caution** ———
> Array `parameterInfoList` must have at least this size.

`parameterInfoList`

> is the array of parameter infos returned. This array must be allocated by the caller.

> ——— **Note** ———
> The minimum size of this array is `desiredNumberOfParameterInfos`.

`actualNumberOfParameterInfos`

> is the actual number of parameter infos returned.

A.3.6   CADISimulationFactory::Instantiate()

This method instantiates and returns a CADI simulation based on the given parameter values. Errors occurring during system initialization are signaled through the given error callback `CADIErrorCallback`.

——— **Note** ———
This call might require a significant amount of time to complete. The call does not return until the instantiation is completed.

```
virtual CADISimulation * CADISimulationFactory::Instantiate(
                    CADIParameterValue_t *parameterValues,
                    CADIErrorCallback *errorCallbackObject,
                    CADISimulationCallback *simulationCallbackObject,
                    char simulationCallbacksEnable[CADI_SIM_CB_Count]) =0;
```

where:

`parameterValues`

> are the parameter values for the simulation as specified by the parameter infos returned by `GetParameterInfos()`.

errorCallbackObject

> is the error callback object to be used for signaling error conditions during simulation.

simulationCallbackObject

> is the callback object to be used for signaling model-wide conditions.

simulationCallbacksEnable

> The elements of this array enable or disable specific simulation callbacks.

> —— **Note** ——

> The simulation must always check if the callbacks are enabled or not and these must not be called if they are disabled. The listener might not want to be called in certain cases.

return value

> is the pointer to the created simulation or NULL if instantiation failed.

A.4   The CADIErrorCallback class

CADIErrorCallback is the base class for error callback handlers addressed during instantiation. The declaration is shown in Example A-6:

Example A-6 The CADIErrorCallback class

```
class CADI_WEXP CADIErrorCallback : public CAInterface
{
public:
    // Return the CAInterface name for this interface.
    static if_name_t IFNAME() { return "eslapi.CADIErrorCallback2"; }
    // Specify the current minor revision for this interface.
    static if_rev_t IFREVISION() { return 0; }
    // This message is called to signal an error to the listeners
    virtual void Error(CADIFactorySeverityCode_t severity,
            CADIFactoryErrorCode_t errorCode, uint32_t erroneousParameterId,
            const char *message) =0;
};
```

A.4.1   CADIErrorCallback::Error()

This method is called to signal an error to the listeners.

```
virtual void Error(CADIFactorySeverityCode_t severity,
                   CADIFactoryErrorCode_t errorCode,
                   uint32_t erroneousParameterId,
                   const char *message) =0;
```

where:

severity   is the severity of the error (see *CADIFactorySeverityCode_t* on page B-3).

errorCode   is the error code as defined in the CADIFactoryErrorCode_t type.

erroneousParameterId

        if this error refers to a parameter, this is the ID of the parameter causing the error.

message   is the error message.

A.5   The CADISimulationCallback class

CADISimulationCallback is the base class for simulation callbacks. It enables registering as a listener for system-wide callbacks.

Example A-7 The CADISimulationCallback class

```
class CADI_WEXP CADISimulationCallback : public CAInterface
{
public:
   // Return the CAInterface name for this interface.
   static if_name_t IFNAME() { return "eslapi.CADISimulationCallback2"; }
    // Specify the current minor revision for this interface.
   static if_rev_t IFREVISION() { return 0; }
   virtual void simMessage(const char *message) = 0;
   virtual void simShutdown() = 0;
   virtual void simKilled() = 0;
};
```

A.5.1   CADISimulationCallback::simMessage()

This method enables the simulation to send system-wide messages to all listeners.

virtual void CADISimulationCallback::simMessage(const char *message) =0;

where:

message   is the message text to send to the listeners.

A.5.2   CADISimulationCallback::simShutdown()

This method enables the simulation to signal that it is shutting down. All clients are requested to unregister their callback handlers, and release any references to the simulation.

virtual void CADISimulationCallback::simShutdown() =0;

A.5.3   CADISimulationCallback::simKilled()

The simulation is being forcedly terminated. After this call returns, the client must cease all communication with the simulation. This callback is intended to provide last-ditch recovery in situations where it is not possible to go through the clean simShutdown() route.

virtual void CADISimulationCallback::simKilled() =0;

A.6 The CADISimulation class

This section describes the CADISimulation class that represents a single simulation. The class has the following methods:

A.6.1 CADISimulation class definition

This CADISimulation class definition is listed in Example A-8.

Example A-8 The CADISimulation class

```
class CADI_WEXP CADISimulation : public CAInterface
{
public:
  static if_name_t IFNAME() { return "eslapi.CADISimulation2"; }
  static if_rev_t IFREVISION() { return 0; }
  virtual void Release(bool shutdown) = 0;
  virtual void AddCallbackObject(CADISimulationCallback *callbackObject) = 0;
  virtual void RemoveCallbackObject(CADISimulationCallback *callbackObject) = 0;
  virtual CADIReturn_t GetTargetInfos(uint32_t startTargetInfoIndex,
                              uint32_t desirednumberOfTargetInfos,
                              CADITargetInfo_t *targetInfoList,
                              uint32_t *actualNumberOfTargetInfos) = 0;
  virtual CAInterface *GetTarget(uint32_t targetID) = 0;
};
```

A.6.2 CADISimulation::IFNAME()

This method returns the CAInterface name for this interface.

```
static if_name_t IFNAME() { return "eslapi.CADISimulation2"; }
```

A.6.3 CADISimulation::IFREVISION()

This method specifies the current minor revision for this interface.

```
static if_rev_t IFREVISION() { return 0; }
```

A.6.4 CADISimulation::Release()

This method releases this simulation and disconnects and cleans-up targets obtained from this simulation. Using a target obtained from a simulation after the simulation is released is illegal.

```
virtual void Release(bool shutdown) =0;
```

where:

shutdown   if `true`, the simulation must call the SystemC method `simulation_quit()` to invoke
            a callback in the SystemC wrapper component and force shutdown and exit.

> ——— **Note** ———
>
> The exit behavior can be overridden by registering for the callback.

A.6.5   CADISimulation::AddCallbackObject()

This method registers to listen for simulation-wide events such as, for example, system
messages.

```
virtual void AddCallbackObject(CADISimulationCallback *callbackObject) =0;
```

where:

`callbackObject`

> is the callback listener to register.

A.6.6   CADISimulation::RemoveCallbackObject()

This method logs off as a listener for simulation-wide events such as, for example, system
messages.

```
virtual void RemoveCallbackObject(CADISimulationCallback *callbackObject) =0;
```

where:

`callbackObject`

> is the callback listener to de-register.

A.6.7   CADISimulation::GetTargetInfos()

This method obtains info about the targets that are provided when the simulation is instantiated.

```
virtual CADIReturn_t GetTargetInfos(uint32_t startTargetInfoIndex,
                                    uint32_t desirednumberOfTargetInfos,
                                    CADITargetInfo_t *targetInfoList,
                                    uint32_t *actualNumberOfTargetInfos) =0;
```

where:

`startTargetInfoIndex`

> is the index of first target info to return. If `startTargetIndex` exceeds the
> maximum target index, `CADI_STATUS_IllegalArgument` is returned.

`desiredNumberOfTargetInfos`

> is the required number of target infos to return.
>
> Array `simulationList` must have at least this size.

`targetInfoList`

> is an array of target informations returned. This array must be allocated by the
> caller. The minimum size of this array is `desiredNumberOfTargetInfos`.

`actualNumberOfTargetInfos`

> is the actual number of target infos returned.

A.6.8   CADISimulation::GetTarget()

This method returns an interface handle for the target with a given target ID. If no CADI exists with id `targetID`, 0 is returned.

```
virtual CAInterface *GetTarget(uint32_t targetID) =0;
```

where:

`targetID`

is the id of the target to return.

A.7 The CADICallbackObj class

This section describes the `CADICallbackObj` class is the base class for the `CADI` callbacks in the component. The class has the following methods:

- *CADICallbackObj::appliOpen()*
- *CADICallbackObj::appliInput()* on page A-17
- *CADICallbackObj::appliOutput()* on page A-17
- *CADICallbackObj::appliClose()* on page A-17
- *CADICallbackObj::doString()* on page A-18
- *CADICallbackObj::modeChange()* on page A-18
- *CADICallbackObj::reset()* on page A-18
- *CADICallbackObj::cycleTick()* on page A-18
- *CADICallbackObj::killInterface()* on page A-19
- *CADICallbackObj::bypass()* on page A-19
- *CADICallbackObj::lookupSymbol()* on page A-19
- *CADICallbackObj::refresh()* on page A-19.

A.7.1 CADICallbackObj class declaration

The class declaration is shown in Example A-9:

Example A-9 The CADICallbackObj class

```
class CADI_WEXP CADICallbackObj : public CAInterface
{
public:
    virtual uint32_t appliOpen(const char *sFileName, const char *mode) = 0;
    virtual void appliOutput(uint32_t streamId, uint32_t count,
                        uint32_t * actualCount, const char *buffer) = 0;
    virtual uint32_t appliClose(uint32_t streamID) = 0;
    virtual void doString(const char *stringArg) = 0;
    virtual void modeChange(uint32_t newMode, CADIBptNumber_t bptNumber) = 0;
    virtual void reset(uint32_t resetLevel) = 0;
    virtual void cycleTick(void) = 0;
    virtual void killInterface(void) = 0;
    virtual uint32_t bypass(uint32_t commandLength, const char * command,
                        uint32_t maxResponseLength, char *response) =0;
    virtual uint32_t lookupSymbol (uint32_t symbolLength, const char *symbol,
                        uint32_t maxResponseLength, char *response) =0;
    virtual void refresh(uint32_t refreshReason) = 0;
};
```

A.7.2 CADICallbackObj::appliOpen()

Opens an application and returns the ID of the stream.

——— **Note** ———

Use of this function is deprecated in CADI 2.0.

```
virtual uint32_t CADICallbackObj::appliOpen(const char * sFileName,
                                            const char * mode) =0;
```

where:

sFileName   is name of the file to be opened.

mode   indicates the permitted access on the file. See the ANSI C definition of fopen for possible values of this parameter.

A.7.3   CADICallbackObj::appliInput()

The target can call this function to request interactive console input from the debugger. The target must call this function only on the first debugger in the list of registered callback objects that implement this function and ignore the callbacks for all following connected debuggers that implement this function. This is in contrast to appliOutput() which is always broadcast to all connected debuggers.

```
virtual void CADICallbackObj::appliInput(uint32_t streamId,
                                         uint32_t count,
                                         uint32_t * actualCount,
                                         char * buffer) =0;
```

where:

streamId   is the stream identifier. This must be set to CADI_STREAMID_STDIN.

count   is the number of characters requested.

actualCount   is the number of characters supplied. This number must never be greater than the number of characters requested. If this number is equal to the number of characters requested, the caller can repeat the call to request more input. A return value of 0 indicates end of file. A return of −1, ~unit32(0), indicates an error such as, for example, an invalid stream ID.

buffer   is the supplied character stream. The buffer is not null terminated.

A.7.4   CADICallbackObj::appliOutput()

Print console output in all connected debuggers that implement this callback function.

```
virtual void CADICallbackObj::appliOutput(uint32_t streamId, uint32_t count,
        uint32_t * actualCount, const char * buffer) =0;
```

where:

streamId   is the stream identifier and must be either CADI_STREAMID_STDOUT or CADI_STREAMID_STDERR.

count   is the number of characters to output.

actualCount   is the number of characters output to the file. A return value of 0 indicates end of file. A return of −1, ~unit32(0), indicates an error.

buffer   contains the characters to output. This buffer can contain NULL characters and is not NULL terminated.

A.7.5   CADICallbackObj::appliClose()

——— **Note** ———

This function is deprecated in CADI 2.0. Do not use it in new models.

Close the stream opened by `appliOpen()`. If the return value is 1, the file was successfully closed. A return value of −1 indicates an error.

```
virtual uint32_t CADICallbackObj::appliClose(uint32_t streamID) =0;
```

A.7.6   CADICallbackObj::doString()

Output a string from the target to the debugger. This can be used, for example, to handle error messages from the target rather than using semihosting to output the message.

```
virtual void CADICallbackObj::doString(char * stringArg) =0;
```

A.7.7   CADICallbackObj::modeChange()

Reports a mode change from the target to the debugger.

```
virtual void CADICallbackObj::modeChange(uint32_t newMode,
                                         CADIBptNumber_t bptNumber) =0;
```

where:

newMode   is one of the `CADI_EXECMODE_*` constants (see *CADI_EXECMODE_t enumeration on page B-26*).

bptNumber   is the breakpoint number. This value is used if the debugger has an action associated with a particular breakpoint. Temporary breakpoints, for example, might run a script after the breakpoint was hit.

This parameter can be ignored for all mode changes not related to a breakpoint.

——— **Note** ———

The `modeChange(CADI_EXECMODE_ResetDone)` callback is identical to the legacy `CADICallbackObj::reset()` callback. See *CADICallbackObj::reset()*.

Targets must support both callbacks to maintain backwards compatibility.

ARM recommends using `modeChange(CADI_EXECMODE_ResetDone)` in client code because the `reset()` callback is deprecated in a future version of CADI.

A.7.8   CADICallbackObj::reset()

Reports a finished target reset to the client.

```
virtual void CADICallbackObj::reset(uint32_t resetLevel) =0;
```

——— **Note** ———

`CADICallbackObj::reset()` is a legacy callback and is identical to the newer `modeChange(CADI_EXECMODE_ResetDone)` callback. See *CADICallbackObj::modeChange()*.

Targets must support both callbacks to maintain backwards compatibility.

ARM recommends using `modeChange(CADI_EXECMODE_ResetDone)` in client code because the `reset()` callback is deprecated in a future version of CADI.

A.7.9   CADICallbackObj::cycleTick()

Deprecated. Do not use.

```
virtual void CADICallbackObj::cycleTick(void) =0;
```

A.7.10 CADICallbackObj::killInterface()

Deprecated. Do not use.

```
virtual void CADICallbackObj::killInterface(void) =0;
```

A.7.11 CADICallbackObj::bypass()

Reserved for future use by the callback object.

```
virtual uint32_t CADICallbackObj::bypass(uint32_t commandLength,
                                         const char * command,
                                         uint32_t maxResponseLength,
                                         char * response) =0;
```

A.7.12 CADICallbackObj::lookupSymbol()

Reserved for future use by the callback object.

```
virtual uint32_t CADICallbackObj::lookupSymbol(uint32_t symbolLength,
                                               const char * symbol,
                                               uint32_t maxResponseLength,
                                               char * response) =0;
```

A.7.13 CADICallbackObj::refresh()

Use this callback whenever the state of a target changes spontaneously while the model is in the stopped state. Do not use it with a modeChange(Stop), modeChange(Error) or modeChange(ResetDone) callback.

A target can notify a debugger to update its display if, for example, a register value changes in the target because it was edited by a debugger. The target uses refresh(REGISTERS) to notify the other debuggers of the register change. If, however, a target hits a breakpoint and stops, it must call the necessary modeChange() callbacks instead of the refresh() callbacks.

```
virtual void CADICallbackObj::refresh(uint32_t refreshReason) =0
```

See *CADIRefreshReason_t* on page B-10 for the definition of the refresh reason constants.

See *CADI_EXECMODE_t enumeration* on page B-26 for details on the relationship between modeChange() callbacks and refresh() callbacks. A target must not call this function while the simulation is running.

A.8 The CADI class

This section describes the `CADI` class and its methods:

- *CADI::CADIExecUnLoadApplication()* on page A-41
- *CADI::CADIExecGetLoadedApplication()* on page A-41
- *CADI::CADIGetInstructionCount()* on page A-42
- *CADI::CADIGetCycleCount()* on page A-42
- *CADI::CADIBptGetList()* on page A-43
- *CADI::CADIBptRead()* on page A-44
- *CADI::CADIBptSet()* on page A-45
- *CADI::CADIBptClear()* on page A-45
- *CADI::CADIBptConfigure()* on page A-45.

A.8.1 Overview of the methods in the CADI class

The methods in the CADI class provide the main interfaces for configuring and running the target:

- *Setup API*
- *Breakpoint API*
- *Execution API*
- *Register API* on page A-22
- *Memory API* on page A-22
- *Cache API* on page A-22.

———— **Note** ————

For details of the structures, enumerations, and defines used with the CADI interface, see the CADITypes.h file and *The CADI class* on page A-20.

Unsupported methods must return CADI_STATUS_CmdNotSupported if called.

———————————

Setup API

The setup API controls the interaction between the host, the debugger, and the CADI target. Use this API to:

- inspect the actual properties of a given CADI object
- register CADICallbackObj callbacks
- bypass specialized commands not available from CADI.

Breakpoint API

The breakpoint API enables defining the following types of breakpoints in the target model:

- instruction execution
- the content of a memory location
- the content of a register
- temporary breakpoints for *run to* debugger behavior
- breakpoints on triggered exceptions.

Execution API

The execution API enables a debugger to:

- control the execution using various asynchronous execution commands
- control the target by, for example, starting or stopping simulation
- obtain information about the pipeline for a cycle-accurate model
- manage the synchronous commands of loading or resetting an application.

Register API

The register API exposes the internal state of the registers of a model for inspection and modification.

If a model has a large number of registers, the registers can be grouped to simplify navigating through the registers. The register API supports compound registers.

Models must expose their internal performance counters (for example, Instr Cache Reads, Instr Cache Misses) as registers to be accessible through this interface.

Memory API

The memory API exposes the internal state of the memory of a model for inspection and modification. Memory is exposed through *address spaces* (memory spaces) that represent separately addressable units.

For processor models, the memory exposed through the API is not memory contained in the model, but rather memory accessed by the model.

Some processor models, however, do contain their own physical memory and expose this memory as a separate memory space.

The requirement for multiple memory spaces is because of different processor models:
* Harvard architectures can require two separate memory spaces
* DSP processors might require up to three memory spaces.
* There also exist processors that access different memory spaces depending on internal execution flags, for instance distinguishing between secure memory and non-secure memory.

Memory models typically expose a single memory space corresponding to their physical memory and other models typically do not expose any memory.

Data stored in a memory space is organized according to the endianness specified by the flags of that particular memory space. This can be little endian or big endian, with the *invariance* defining the number of bytes in an accessed unit.

Data can also be organized using a model-specific endianness. In these cases, the documentation that accompanies the model must provide specific details.

The total number of bytes in a memory word can be determined based on `bitsPerMau`. The bytes are divided in groups of `invariance` bytes. These groups are then arranged in little endian or big endian order.

For example, for `invariance` of 2 and `bitsPerMau` of 64:
* a little endian word is represented as `b0 b1 b2 b3 b4 b5 b6 b7`
* a big endian word is represented as `b6 b7 b4 b5 b2 b3 b0 b1`.

Each memory space can be subdivided in memory blocks. Memory blocks contain additional information pertaining to the intended usage of the memory. This information can be used as hints for memory data presentation dedicated for human consumption, but it has no effect on the actual simulation.

Cache API

These functions enable access to cache memories in the target. Use the `CADIGetCacheInfo()` function to return the cache information for the target. The `CADICacheRead()` and `CADICacheWrite()` functions are used to directly access the cache memory contents.

Parameters API

The parameters API enables:

- getting information on runtime parameters
- retrieving the current values for runtime parameters
- setting runtime parameters to new values.

A.8.2 The component CADI class declaration

The class declaration can typically be left as in Example A-10 with the exception of:

- adding any private data members
- changing the parameter in the constructor to the class name of the component.

——— **Note** ———

If your component is a processor, see also the functions that are available in CADIDisassembler and CADIProfiler classes for controlling and monitoring application execution. For more information, see:

Example A-10 Header file for a typical CADI component class

```
class CADIMyComponent : public CADI
{
public:
    CADIMyComponent(MyComponentClass* c); // Change names accordingly
    virtual ~CADIMyComponent();
// The declaration/implementation of CAInterface() methods is missing
// and must be added at this point.
// These are essential for properly obtaining a CADI 2.0 interface
public:
    // Register access functions
    CADIReturn_t    CADIRegGetGroups(uint32_t groupIndex
        , uint32_t desiredNumOfRegGroups, uint32_t* actualNumOfRegGroups
        , CADIRegGroup_t* reg);
    CADIReturn_t    CADIRegGetMap( uint32_t groupID, uint32_t regIndex
        , uint32_t registerSlots, uint32_t* registerCount, CADIRegInfo_t* reg);
    CADIReturn_t    CADIRegWrite( uint32_t regCount, CADIReg_t* reg
        , uint32_t* numRegsWritten, uint8_t doSideEffects);
    CADIReturn_t    CADIRegRead(uint32_t regCount, CADIReg_t* reg
        , uint32_t* numRegsRead, uint8_t doSideEffects);
    // Memory access functions
    CADIReturn_t    CADIMemGetSpaces( uint32_t spaceIndex, uint32_t memSpaceSlots
        , uint32_t* memSpaceCount, CADIMemSpaceInfo_t* memSpace);
    CADIReturn_t    CADIMemGetBlocks( uint32_t memorySpace, uint32_t blockIndex
        , uint32_t memBlockSlots, uint32_t* memBlockCount
        , CADIMemBlockInfo_t* memBlock);
    CADIReturn_t    CADIMemWrite( CADIAddrComplete_t startAddress
        , uint32_t unitsToWrite, uint32_t unitSizeInBytes, const uint8_t *data
        , uint32_t *actualNumOfUnitsWritten, uint8_t doSideEffects);
    CADIReturn_t    CADIMemRead( CADIAddrComplete_t startAddress
        , uint32_t unitsToRead, uint32_t unitSizeInBytes, uint8_t *data
        , uint32_t *actualNumOfUnitsRead, uint8_t doSideEffects);
    // Access to disassembly class (if available)
```

```
            CADIDisassembler* CADIGetDisassembler(void);
    private:
        // Pointer to your own component class, see constructor
        MyComponentClass *          target;
        // Register related info
        CADIRegInfo_t*              regInfo;
        CADIRegGroup_t*             regGroup;
        // Memory related info
        CADIMemSpaceInfo_t*         memSpaceInfo;
        CADIMemBlockInfo_t*         memBlockInfo;
    };
```

A.8.3 The CADI class constructor

You can define in the constructor the number of registers you have and the property of your memory spaces.

A.8.4 CADI::CADIXfaceGetFeatures()

The debugger for a target must call this function when it attaches to a target. This function is typically called once per target. The debugger can, however, call it more often if required. This call determines the features supported by the target by updating the passed `features` parameter.

```
virtual CADIReturn_t CADI::CADIXfaceGetFeatures(
                                CADITargetFeatures_t * features) =0;
```

The caller allocates and de-allocates memory for the *features* parameter. For details on the `CADITargetFeatures_t` structure, see *CADITargetFeatures_t structure* on page B-5.

A.8.5 CADI::CADIXfaceGetError()

If an error is detected, this routine is called to get the error message.

```
virtual CADIReturn_t CADI::CADIXfaceGetError(uint32_t maxMessageLength,
                                uint32_t * actualMessageLength,
                                char * errorMessage) =0;
```

where:

maxMessageLength

> is the max length of `errorMessage` array. The target must not fill more than this number of characters in the array.

actualMessageLength

> is the max length of `errorMessage` array. The target must set this to the actual number of chars written into the `errorMessage` buffer.

errorMessage is the actual error message text. The target writes the text into this character buffer. The length of this buffer is exactly `maxMessageLength`.

A.8.6 CADI::CADIGetDisassembler()

——— **Caution** ———

Obtaining a disassembler from the CADI interface by calling `CADIGetDissambler()` is deprecated. The function is retains to enable compatibility with CADI 1.1.

New code must use the `ObtainInterface()` call for both disassembler and profiling support.

This method returns the `CADIDisassembler` for a target.

```
virtual CADIDisassembler * CADI::CADIGetDisassembler(void) =0;
```

A.8.7   CADI::CADIXfaceAddCallback()

A debugger connected to the target must call this to register a callback object that handles asynchronous information from the target. The callback routines must not make calls to the target. It is possible for a debugger to receive a callback while in the middle of a call by, for example, receiving a `modeChange` callback from within a `CADIExecStop` call.

Callbacks from a target into the debugger typically come from a different thread (called the simulation thread) than the calls from the debugger into the target (called the GUI thread or debugger thread).

Already registered callbacks can be reconfigured with respect to the enabled callbacks. That is, they are replaced when called again.

```
virtual CADIReturn_t CADI::CADIXfaceAddCallback(CADICallbackObj * callbackObj,
                                     char enable[CADI_CB_Count]) =0;
```

where:

callbackObj   is a pointer to the object whose member functions are called as callbacks.

enable   the elements of this array enable or disable specific callbacks. The caller must always check if the callbacks are enabled. The callbacks must not be called if they are disabled.

              The indexes in the array must be based on the list in `CADICallbackType_t`. The length of the array is `CADI_CB_Count`.

A.8.8   CADI::CADIXfaceRemoveCallback()

A debugger must call this to remove any callback objects it has added. This is required when disconnecting from a target that is not shut down.

```
virtual CADIReturn_t CADI::CADIXfaceRemoveCallback(
                                     CADICallbackObj * callbackObj) =0;
```

where:

callbackObj   is a pointer to the callback object. The target must not use this object after this call.

A.8.9   CADI::CADIXfaceBypass()

Targets can have specialized commands that can be requested by the debugger. This command enables the debugger to pass a string containing one of these commands to a target. The target must silently ignore all unknown commands issued through this mechanism and on return set `response` to an empty string and use `CADI_STATUS_UnknownCommand` as the return value.

```
virtual CADIReturn_t CADI::CADIXfaceBypass(uint32_t commandLength,
                                     const char * command,
                                     uint32_t maxResponseLength,
                                     char * response) =0;
```

where:

commandLength

>is the length, including the terminating zero, of the command. This helps networked versions of the interface to determine how much space to allocate for command.

command   is the entire command with all arguments.

maxResponseLength

>is the length of the response array. The target must truncate the response to fit it into the array.

response   is the response from the target. This string might or might not be zero terminated. It might also be `Null` or contain binary data depending on the issued bypass commands

A.8.10 CADI::CADIGetTargetInfo()

Return target information for this model. The values for the return parameters are set by the model.

```
virtual CADIReturn_t CADI::CADIGetTargetInfo(CADITargetInfo_t *targetInfo) =0;
```

where:

targetInfo   is set to point to the `CADITargetInfo_t` structure.

A.8.11 CADI::CADIGetParameterInfo()

Get parameter info class for a specific parameter name.

```
virtual CADIReturn_t CADIGetParameterInfo(const char *parameterName,
                           CADIParameterInfo_t *param) =0;
```

where:

parameterName

>is the name of the parameter to be retrieved. This is the local name in the model, not the global hierarchical name.

param   points to a single `CADIParameterInfo_t` buffer that must be pre-initialized by the caller and filled with data by the callee.

A.8.12 CADI::CADIGetParameterValues()

Return the current parameter values.

```
virtual CADIReturn_t CADI::CADIGetParameterValues(uint32_t parameterCount,
                           uint32_t *actualNumOfParamsRead,
                           CADIParameterValue_t *paramValuesOut) =0;
```

where:

parameterCount

>is the length of array `paramValuesOut`.

actualNumOfParamsRead

> is the number of valid entries in `paramValuesOut`. ARM recommends that this is initialized to 0 by the caller.
>
> If an error code is returned and `actualNumOfParamsRead` is greater than 0, the first `actualNumOfParams` entries are valid and caused no error. The entry `paramValuesOut[actualNumOfParamsRead]` caused the error.

paramValuesOut

> is an output buffer that holds the parameter values.

A.8.13 CADI::CADIGetParameters()

Get list of supported parameters and parameter details.

```
virtual CADIReturn_t CADI::CADIGetParameters(uint32_t startIndex,
                                             uint32_t desiredNumOfParams,
                                             uint32_t * actualNumOfParams,
                                             CADIParameterInfo_t * params) =0;
```

A.8.14 CADI::CADISetParameters()

Set parameter values.

```
virtual CADIReturn_t CADI::CADISetParameters(uint32_t parameterCount,
                                             CADIParameterValue_t * parameters,
                                             CADIFactoryErrorMessage_t * error) =0;
```

A.8.15 CADI::CADIRegGetGroups()

This call is used to retrieve register groups from the target.

```
virtual CADIReturn_t CADI::CADIRegGetGroups(uint32_t groupIndex,
                                            uint32_t desiredNumOfRegGroups,
                                            uint32_t * actualNumOfRegGroups,
                                            CADIRegGroup_t * reg) =0;
```

where:

groupIndex   is the index into the internal list of register groups as maintained by the target. It is *not* the group IDs.

desiredNumOfRegGroups

> is the size of the `reg[]` buffer provided by the caller.

actualNumOfRegGroups

> is, on return, the number of groups that have actually been returned by the target. If this is less than the number requested, the debugger might call this function again with a different `groupIndex`. Any value set on input is ignored.

reg   is the register group information. The array is allocated, and deallocated if applicable, by the caller and filled by the target:

- The amount of space allocated must be enough to hold the number of groups required.

- If the required count is greater than the targets total number of register groups, the target must return all groups.

- If fewer groups are returned than requested, the last entries of the `reg[]` array are left empty.

A.8.16 CADI::CADIRegGetMap()

It is recommended that the debugger for the target call this method after connecting to the target to obtain detailed register information:

- All registers must be reported even if they are part of a compound register.

- All register numbers must be unique both for registers in the same group and register numbers in other groups.

- A register can be a member of more than one register group.

```
virtual CADIReturn_t CADI::CADIRegGetMap(uint32_t groupID,
                                         uint32_t startRegisterIndex,
                                         uint32_t desiredNumOfRegisters,
                                         uint32_t * actualNumOfRegisters,
                                         CADIRegInfo_t * reg) =0;
```

where:

groupID   identifies the ID of the group whose map is requested. If the value is `CADI_REG_ALLGROUPS`, all registers of all groups are returned.

startRegisterIndex

is the index into the internal list of registers held by the target. It is *not* register numbers.

desiredNumOfRegisters

is the total number of registers required by the caller. The caller must allocate a buffer size that is enough to hold the requested number of registers.

actualNumOfRegisters

is the number of registers actually returned by the target. Any value set on input is ignored.

reg   is the register information. The array is allocated, and deallocated if applicable, by the caller to be filled by the target. The amount of space allocated must be enough to hold the number of registers requested.

If the count is greater than the targets number of registers, the target must return all the registers. If fewer registers are returned than requested, the last entries of `reg[]` are left empty.

A.8.17 CADI::CADIRegGetCompound()

This call gets the information about a compound register. The structure of the compound register is as reported by a call to `CADIRegGetCompound()`. Compound registers, that is, registers that contain subregisters, form an additional hierarchy below register groups.

```
virtual CADIReturn_t CADI::CADIRegGetCompound(uint32_t reg,
                                              uint32_t componentIndex,
                                              uint32_t desiredNumOfComponents,
                                              uint32_t * actualNumOfcomponents,
                                              uint32_t * components) =0;
```

where:

reg   is the register number.

componentIndex

is the index into the internal component array for the requested register.

desiredNumOfComponents

> is the total number of child registers required by the caller, starting at `componentIndex`.

actualNumOfcomponents

> on return, is the number of components returned by the target. Any value set on input is ignored.

components   on return, is the list of component registers. The array is allocated, and deallocated if applicable, by the caller to be filled by the target. The amount of space allocated must be big enough to hold the number of requested components. If a target has written less than `regCount` registers it returns the number of registers successfully written in this field. The target must report an error only in the case of a cyclic graph where, for example, a compound register contains a register (component) that again is a compound register that owns a component that is the initially requested compound register.

A.8.18   CADI::CADIRegWrite()

This function writes to registers in the target.

```
virtual CADIReturn_t CADI::CADIRegWrite(uint32_t regCount,
                                        CADIReg_t * reg,
                                        uint32_t * numOfRegsWritten,
                                        uint8_t doSideEffects) =0;
```

where:

regCount   is the requested number of registers (and consequently the size of the reg array).

reg   is an array of `CADIReg_t` structures each holding the some attributes and an array of bytes containing the contents of an individual register. The number of required bytes for each register is available from the `CADIInfo_t` structure. The number of registers is returned by the `CADIRegGetMap()` call.

numOfRegsWritten

> on return, is the number of registers that are actually written. Any value set on input is ignored.

doSideEffects

> If set to `true`, this parameter informs the target that it must perform side effects on a write access. Such side effects might be, for example, triggering an interrupt. If it is set to `false`, the target must decide when to ignore this parameter. For some cases it is not possible to write a register without doing a side effect such as manipulating a register that influences a hardware accelerator's behavior and changes the computed results.

A.8.19   CADI::CADIRegRead()

This function reads register values from the target.

```
virtual CADIReturn_t CADI::CADIRegRead(uint32_t regCount,
                                       CADIReg_t * reg,
                                       uint32_t * numRegsRead,
                                       uint8_t doSideEffects) =0;
```

where:

regCount   is the number of requested registers and consequently the size of the reg array.

reg   is an array of `CADIReg_t` structures each holding the some attributes and an array of bytes containing the contents of an individual register. The number of required bytes for each register is available from the `CADIInfo_t` structure. The number of registers is returned by the `CADIRegGetMap()` call..

numRegsRead   on return, is the number of registers actually read. If the value is less than `regCount`, the function returns an error code. Any value set on input is ignored.

doSideEffects

if this parameter is set to `true`, it informs the target that it must perform side effects on a read access. Such side effects might be, for example, a clear-on-read.

If the parameter is set to `false`, the target must always omit side effects. This is the common use case where a debug read of a register must not interfere with the target execution.

——— **Note** ———

If an error occurs, `CADIRegRead()` must return the error position in `numRegsRead`. Data is assumed valid up to this position.

The code in Example A-11 shows how to set up the CADI access functions and test reading a register value:

Example A-11 CADI register access

```
CADI * cadi1 = s1->getCADI(); // cadi1 is a valid CADI interface
uint32_t actual =0;
CADIRegGroup_t regGroups [2];
cadi1->CADIRegGetGroups(0, 2, & actual, regGroups);
CADIRegInfo_t regs [2];
actual =0;
cadi1->CADIRegGetMap(regGroups [0].groupID, 0, 2, & actual, regs);
CADIReg_t reg;
reg.regNumber = regs [1].regNumber;
actual =0;
cadi1->CADIRegRead(1, & reg, & actual, 0);
printf("CADI reg 0x%x\n", reg.bytes [0]);
```

A.8.20   CADI::CADIGetPC()

Returns the PC of the instruction that is executed next from an ISA perspective.

```
virtual uint64_t CADI::CADIGetPC() =0;
virtual uint64_t CADI::CADIGetPC(bool * is_virtual) =0;
```

A.8.21   CADI::CADIGetCommittedPCs()

The function returns the number of program counters in the current cycle. This can be used with multi-issue processors.

```
virtual CADIReturn_t CADIGetCommitedPCs(int startIndex,
                                        int desiredCount,
                                        int * actualCount,
                                        uint64_t * pcs) =0;
```

where:

startIndex   is the index into the internal buffer of PCs present in the target.

desiredCount is the required number of PCs.

actualCount   is the total number of PCs returned by the target through the pcs[] array.

pcs   is a list of PCs. The array is allocated, and deallocated if applicable, by the caller
              to be filled by the target. This space must be big enough to hold the required
              number of spaces.

A.8.22 CADI::CADIMemGetSpaces()

It is recommended that the debugger call this after connecting to the target but before accessing
any memory. The function identifies the number of independent address spaces available on the
target. Use different memory spaces to separate distinct memory areas with overlapping address
values (like program and data memory in a Harvard architecture).

```
virtual CADIReturn_t CADI::CADIMemGetSpaces(uint32_t startMemSpaceIndex,
                                            uint32_t desiredNumOfMemSpaces,
                                            uint32_t * actualNumOfMemSpaces,
                                            CADIMemSpaceInfo_t * memSpaces) =0;
```

where:

startMemSpaceIndex

              is the index into the buffer of memory spaces present in the target.

desiredNumOfMemSpaces

              is the required number of memory spaces.

actualNumOfMemSpaces

              is the total number of memory spaces returned by the target.

memSpaces   is a list of memory spaces. The array is allocated, and deallocated if applicable,
              by the caller to be filled by the target. This space must be big enough to hold the
              required number of spaces.

A.8.23 CADI::CADIMemGetBlocks()

It is recommended that the debugger for the target call this once for each memory space,
provided by the calling the CADIMemGetSpaces() function, before accessing memory in that space.
This must return the layout of the memory in a specific block. No two blocks with the same
parent can overlap. This call returns existing memory blocks only. The caller can assume that
any memory that is not in a block is a gap or invalid memory.

```
virtual CADIReturn_t CADI::CADIMemGetBlocks(uint32_t memorySpace,
                                            uint32_t memBlockIndex,
                                            uint32_t desiredNumOfMemBlocks,
                                            uint32_t * actualNumOfMemBlocks,
                                            CADIMemBlockInfo_t * memBlocks) =0;
```

where:

memorySpace   is the ID of the memory space for which the caller requests a block list.

memBlockIndex

              is the index into the internal buffer of memory blocks held by the target for the
              specified memory space.

desiredNumOfMemBlocks

> is the required number of memory blocks.

actualNumOfMemBlocks

> is the is the total number of blocks returned by the target. It is less than the number requested if the number requested is more than the number available.

memBlocks   is a buffer that must be big enough to hold the required number of CADIMemBlockInfo_t structures. Space is allocated, and deallocated if applicable, by the caller.

A.8.24   CADI::CADIMemRead()

The function reads memory values from the component. This function must be implemented to support the display of memory contents.

```
virtual CADIReturn_t CADI::CADIMemRead(CADIAddrComplete_t startAddress,
                                       uint32_t unitsToRead,
                                       uint32_t unitSizeInBytes,
                                       uint8_t * data,
                                       uint32_t * actualNumOfUnitsRead,
                                       uint8_t doSideEffects) =0;
```

where:

startAddress is the starting address to begin reading from. If startAddress.overlay is CADI_NO_OVERLAY, it refers to the current overlay.

unitsToRead   is the number of units of size unitSizeInBytes to read.

unitSizeInBytes

> is the unit size, specified in bytes, for memory accesses.

data   is the data buffer that was allocated by the caller and must be big enough to hold the requested number of addresses. The target data is encoded in little endian format.

actualNumOfUnitsRead

> is the number of units actually read. It can be less than the number of units requested.

doSideEffects

> if this parameter is set to true, it informs the target that it must perform side effects on a read access. Such side effects might be, for example, a clear-on-read.
>
> If the parameter is set to false, the target must always omit side effects. This is the common use case where a debug read of memory must not interfere with the target execution.

——— **Note** ———

If an error occurs, CADIMemRead() must return the error position in actualNumOfUnits*. Data is assumed valid up to this position.

A.8.25   CADI::CADIMemWrite()

This function writes values to the memory in the target.

```
virtual CADIReturn_t CADI::CADIMemWrite(CADIAddrComplete_t startAddress,
                                        uint32_t unitsToWrite,
                                        uint32_t unitSizeInBytes,
                                        const uint8_t * data,
                                        uint32_t * actualNumOfUnitsWritten,
                                        uint8_t doSideEffects) =0;
```

where:

startAddress is the starting address to begin writing from. If startAddress.overlay is
CADI_NO_OVERLAY, it refers to the current overlay.

unitsToWrite

is the number of units of size unitSizeInBytes to write.

unitSizeInBytes

is the unit size, specified in bytes, of the memory accesses.

data   is the data buffer holding the values to be written. This contains target data,
            encoded in little endian format.

actualNumOfUnitsWritten

is the number of units actually written to the target. It can be less than the number
of units requested.

doSideEffects

If set to true, this parameter informs the target that it must perform side effects on
a write access. Such side effects might be, for example, triggering an interrupt.

If set to false, the target must decide when to ignore this parameter. For some
cases it is not possible to write to memory without doing a side effect such as
manipulating a memory-mapped register that influences a hardware accelerator's
behavior and changes the computed results.

—— **Note** ——

On error, CADIMemWrite() must return the error position in actualNumOfUnits*. Data is assumed
valid up to this position.

If the write spans a gap in the memory space, the target must stop writing a the beginning of the
gap and return the number of successful writes in numUnitsWritten.

A.8.26 CADI::CADIMemGetOverlays()

The debugger calls this function to get the list of active overlays. This would typically be done
when a breakpoint is hit. When overlays are implemented, an overlay ID must be stored in the
symbol table and in the target software. The symbol table must store the starting address,
memory space, and byte count for each overlay. This enables the ID to be sent to the host when
an overlay occurs.

```
virtual CADIReturn_t CADI::CADIMemGetOverlays(uint32_t activeOverlayIndex,
                                              uint32_t desiredNumOfActiveOverlays,
                                              uint32_t* actualNumOfActiveOverlays,
                                              CADIOverlayId_t * overlays) =0;
```

where:

activeOverlayIndex

is the start index into the internal buffer of overlays held by the target.

desiredNumOfActiveOverlays

is the required number of overlays.

actualNumOfActiveOverlays

is the number of overlay structures returned by the target.

overlays   is the list of overlays that are currently memory resident (that is, swapped-in). The array is allocated, and deallocated if applicable, by the caller and filled by the target.

A.8.27   CADI::VirtualToPhysical()

This function translates the virtual address passed as a parameter to a physical address that is the return value.

```
virtual CADIAddrComplete_t CADI::VirtualToPhysical(CADIAddrComplete_t vaddr) =0;
```

where:

vaddr   is the virtual address that is to be converted.

——— **Note** ———

If the call fails or is not supported, the returned CADIADDRComplete_t has a memory space ID of CADI_MEM_SPACE_NOTSUPPORTED.

A.8.28   CADI::PhysicalToVirtual()

This function translates the physical address passed as a parameter to a virtual address that is the return value.

```
virtual CADIAddrComplete_t CADI::PhysicalToVirtual(CADIAddrComplete_t paddr) =0;
```

where:

paddr   is the physical address that is to be converted.

——— **Note** ———

If the call fails or is not supported, the returned CADIADDRComplete_t has a memory space ID of CADI_MEM_SPACE_NOTSUPPORTED.

A.8.29   CADI::CADIGetCacheInfo()

This call gets the cache information for a memory space.

```
virtual CADIReturn_t CADI::CADIGetCacheInfo(uint32_t memSpaceID,
                                          CADICacheInfo_t * cacheInfo) =0;
```

where:
memSpaceID   is the memory space.
cacheInfo   is the cache information.

A.8.30   CADI::CADICacheRead()

This function performs a cache read.

```
virtual CADIReturn_t CADI::CADICacheRead(CADIAddr_t addr,
                                         uint32_t linesToRead,
                                         uint8_t * data,
                                         uint8_t * tags,
                                         bool * is_dirty,
                                         bool * is_valid,
                                         uint32_t * numLinesRead,
                                         bool doSideEffects) =0;
```

where:

addr   is the address to be read, including the memory space ID.

linesToRead

      is the number of cache lines to read.

data   is a byte array of size (cache_lines * line_size). The array is encoded in little endian format.

tags   is a byte array of size (cache_lines * tagsbits/8).

is_dirty   is the status (one per line).

is_valid   is the status (one per line).

numLinesRead

      is the number of cache lines actually read.

doSideEffects

      If set to true, this parameter informs the target that it must perform side effects on a cache read access. Such side effects might be, for example, triggering an interrupt. If it is set to false, the target must decide when to ignore this parameter. For some cases it is not possible to read from cache without side effects.

A.8.31   CADI::CADICacheWrite()

This function performs a cache write.

```
virtual CADIReturn_t CADI::CADICacheWrite(CADIAddr_t addr,
                                          uint32_t linesToWrite,
                                          const uint8_t * data,
                                          const uint8_t * tags,
                                          const bool * is_dirty,
                                          const bool * is_valid,
                                          uint32_t * numLinesWritten,
                                          bool doSideEffects) =0;
```

where:

addr   is the address to be written, including the memory space ID.

linesToWrite

      is the number of cache lines to write.

data   is a byte array of size (cache_lines * line_size). The array is encoded in little endian format.

tags   is a byte array of size (cache_lines * tagsbits/8).

is_dirty   is status (one per line).

is_valid   is status (one per line).

numLinesWritten

>   is the number of cache lines actually written.

doSideEffects

>   If set to `true`, this parameter informs the target that it must perform side effects such as, for example, selecting write through on a write access. If it is set to `false`, the target must decide when to ignore this parameter. For some cases it is not possible to access cache without side effects.

A.8.32   About the CADI execution modes

The execution APIs modify the execution state of the target.

These functions are asynchronous and typically return before the target completes the requested action. For example, a run or even a single step returns before the target stops. The debugger is notified by the callback about the completion of the request.

The exec mode calls enable extensions to the typical execution modes such as `run`, `stop`, and `breakpoint`. If a target does not have other modes, these calls are redundant and are typically not used.

Execution modes such as `run`, `stop`, and `breakpoint` are associated with specific identifiers as described in *CADI_EXECMODE_t enumeration* on page B-26.

A.8.33   CADI::CADIExecGetModes()

Many processors have more than `run`, `stop`, and `breakpoint` states. This call enables the debugger to determine the additional states.

```
virtual CADIReturn_t CADI::CADIExecGetModes(uint32_t startModeIndex,
                                            uint32_t desiredNumOfModes,
                                            uint32_t * actualNumOfModes,
                                            CADIExecMode_t * execModes) =0;
```

where:

startModeIndex

>   is the index into the internal buffer of execution modes held by the target.

desiredNumOfModes

>   is the requested number of modes.

actualNumOfModes

>   is the number of modes returned by the target.

execModes   is a list of `CADIExecMode_t` structs to receive the requested execution modes. The caller allocates (and, if applicable, deallocates) space. The number of elements must be the same as `desiredNumOfModes` to provide enough space for the requested modes. The mode values are listed in *CADI_EXECMODE_t enumeration* on page B-26.

A.8.34   CADI::CADIExecGetResetLevels()

Many targets have more than one reset level. This call enables the debugger to determine what these levels are.

```
virtual CADIReturn_t CADI::CADIExecGetResetLevels(
                                    uint32_t startResetLevelIndex,
                                    uint32_t desiredNumOfResetLevels,
                                    uint32_t * actualNumOfResetLevels,
                                    CADIResetLevel_t * resetLevels) =0;
```

where:

startResetLevelIndex

> is the index into the internal buffer of reset levels held by the target.

desiredNumOfResetLevels

> is the number of levels required by the caller.

actualNumOfResetLevels

> is the number of reset levels actually returned.

resetLevels is the caller allocated list that receives the requested reset levels. The number of elements must be the same as the desiredNumOfResetLevels to provide space for the requested reset levels. The contents must be returned sorted in order of most severe (at reset level zero) to least severe.

A.8.35 CADI::CADIExecSetMode()

This sets the target to a specified execution mode. This call returns immediately, possibly before the target execution mode has been reached. The mode values are listed in *CADI_EXECMODE_t enumeration* on page B-26

```
virtual CADIReturn_t CADI::CADIExecSetMode(uint32_t execMode) =0;
```

This call is, for a subset of the execution modes, redundant with other APIs:

- A call to CADIExecSetMode(CADI_EXECMODE_Run) is equivalent to a call to CADIExecContinue().

- A call to CADIExecSetMode(CADI_EXECMODE_Stop) is equivalent to a call to CADIExecStop().

--- **Note** ---
execMode must be less than the value nrExecModes received by CADIXfaceGetFeatures().

A.8.36 CADI::CADIExecGetMode()

This call enables the debugger to determine the execution mode of the target.

```
virtual CADIReturn_t CADI::CADIExecGetMode(uint32_t * execMode) =0;
```

--- **Note** ---
execMode must be less than the value nrExecModes received by CADIXfaceGetFeatures().

A.8.37 CADI::CADIExecSingleStep()

This function returns immediately and a separate notification informs the debugger that the execution state has changed. Typically this call results in the modeChange() callback (if enabled) for CADI_EXECMODE_Run followed by CADI_EXECMODE_Stop.

```
virtual CADIReturn_t CADI::CADIExecSingleStep(uint32_t instructionCount,
                                              int8_t stepCycle,
                                              int8_t stepOver) =0;
```

where:

`instructionCount`

> is the number of instructions to be executed.
>
> Some targets can not step a specific number of instructions safely (into a delay slot, for example). In this case, the target can step additional instructions to enable it to stop at a safe place.

`stepCycle` specifies (for targets that have exposed multiple pipe stages) whether the step merely clocks the device (`stepCycle == yes`) or flushes the pipe (`stepCycle == no`).

> For other kinds of targets, this argument is ignored (`stepCycle = no` is assumed).

`stepOver` enables the target to handle stepping over a call.

> It is especially useful for an emulator with no available breakpoints. In this case the target must step until the call returns or a breakpoint is hit.

——— **Note** ———

Because this call returns immediately, the return value indicates whether the target believes that it can perform the operation and not whether the operation was completed successfully.

A.8.38 CADI::CADIExecReset()

On receipt of this call, the target:

* resets its execution related internal state
* resets its registers to their initial state
* does not change breakpoints or callbacks.

This call provides a simulation level reset.

```
virtual CADIReturn_t CADI::CADIExecReset(uint32_t resetLevel) =0;
```

where *resetLevel* must be one of the numbers provided in the `resetLevels` array received by `CADIExecGetResetLevels()`.

On receiving this call, the target reset its execution related internal state. It resets registers and memories to a predefined state, but does not change breakpoints or callbacks.

This call provides a generic reset interface that is independent of the actual model implementation. For example, a debugger can use this call to reset the simulation of a model, system, or subsystem that does not implement an explicit simulation-level reset mechanism.

The list of reset levels is target specific:

* Reset level 0 has fixed semantics and must be implemented by every target. This reset level brings the simulation platform back into the same state as immediately after instantiation. This state must be known to enable deterministic behavior of the platform after each reset.

* All reset levels other than 0 are model specific. The reset levels supported depend on the model implementation.

CADIExecReset() is an asynchronous call. After the reset finishes, the target sends a modeChange(CADI_EXECMODE_ResetDone) callback to all connected debuggers. The reset might be finished at the time that CADIExecReset() returns.

A.8.39 CADI::CADIExecContinue()

This function returns immediately and a separate notification from the modeChange(CADI_EXECMODE_Run) callback informs the debugger when the execution state has changed. The simulation runs asynchronously in a separate thread.

```
virtual CADIReturn_t CADI::CADIExecContinue(void) =0;
```

― **Note** ―

Because this call returns immediately, the return value indicates whether the target believes that it can perform the operation and not whether the operation was completed successfully.

A.8.40 CADI::CADIExecStop()

This causes the execution of the target to stop. The function returns immediately and the target might still be running when the function returns. A debugger must wait for a modeChange(CADI_EXECMODE_Stop) callback to ensure that the simulation has ended.

```
virtual CADIReturn_t CADI::CADIExecStop(void) =0;
```

― **Note** ―

Because this call returns immediately, the return value indicates whether the target believes that it can perform the operation and not whether the operation was completed successfully.

A.8.41 CADI::CADIExecGetExceptions()

This gets the list of the exception vectors for the target.

```
virtual CADIReturn_t CADI::CADIExecGetExceptions(uint32_t startExceptionIndex,
                                  uint32_t desiredNumOfExceptions,
                                  uint32_t * actualNumOfExceptions,
                                  CADIException_t * exceptions) =0;
```

where:

startExceptionIndex

is the index into the targets list of exceptions.

desiredNumOfExceptions

is the number of slots in the exception array. The target must not fill more than this number of characters in the array.

actualNumOfExceptions

is the total number of returned exceptions.

exceptions   is list of exceptions. The array is allocated, and deallocated if applicable, by the caller to be filled by the target. This buffer must be big enough to hold desiredNumOfExceptions.

A.8.42 CADI::CADIExecAssertException()

Raise an exception.

```
virtual CADIReturn_t CADI::CADIExecAssertException(uint32_t exception,
                                    CADIExceptionAction_t action) =0;
```

where:

exception   is the exception number.

action   is the exception action to be taken as described in *CADIExceptionAction_t* on
             page B-29.

A.8.43 CADI::CADIExecGetPipeStages()

This is used to expose the pipeline stages simulated inside of a cycle-accurate simulation.

```
virtual CADIReturn_t CADI::CADIExecGetPipeStages(uint32_t startPipeStageIndex,
                                    uint32_t desiredNumOfPipeStages,
                                    uint32_t * actualNumOfPipeStages,
                                    CADIPipeStage_t * pipeStages) =0;
```

where:

startPipeStageIndex

is the index into the internal list of pipeline stages held by the target.

desiredNumOfPipeStages

is the number of entries to fill in the pipeStages array. The target must not fill
more than this number of elements.

actualNumOfPipeStages

is the number of stages actually returned to the caller.

pipeStages   is the list of pipe stages in order of execution for a single instruction. pipestage[0]
             must contain the first stage executed for any single instruction. The array is
             allocated, and deallocated if applicable, by the caller to be filled by the target.

A.8.44 CADI::CADIExecGetPipeStageFields()

This is used to expose the fields of the pipe simulated inside of a cycle-accurate simulation.

```
virtual CADIReturn_t CADI::CADIExecGetPipeStageFields(
                                    uint32_t startPipeStageFieldIndex,
                                    uint32_t desiredNumOfPipeStageFields,
                                    uint32_t * actualNumOfPipeStageFields,
                                    CADIPipeStageField_t * pipeStageFields) =0;
```

where:

startPipeStageFieldIndex

is the index into the internal list of pipe stage fields held by the target.

desiredNumOfPipeStageFields

is the number of entries to fill in the pipeStageFields array. The target must not
fill more than this number of elements.

actualNumOfPipeStageFields

is the number of stages actually returned to the caller.

pipeStageFields

> is the list of pipe stage fields in order of execution for a single instruction. The list can be sorted, but this is not mandatory. The array is allocated, and deallocated if applicable, by the caller to be filled by the target.

A.8.45 CADI::CADIExecLoadApplication()

This is used to load an application file to program memory. The target is not reset or restarted. The implementation of the model determines which file formats, ELF for example, are supported. The debugger is responsible for initiating the execution of the application by, for example, setting the program counter to the entry point in the application.

```
virtual CADIReturn_t CADI::CADIExecLoadApplication(const char *filename,
                                                   bool loadData,
                                                   bool verbose,
                                                   const char *parameters) =0;
```

where:

filename   is the name of the application file.

loadData   If set to `true`, the target loads data, symbols, and code.

> If set to `false`, the target does not reload the application code to its program memory. This can be used, for example, to either:
> - forward information about applications that are loaded to a target by other platform components
> - change command line parameters for an application that was loaded by a previous `CADIExecLoadApplication()` call.

verbose   If `true`, the target can print verbose messages while loading a file.

> The target decides whether or not it outputs messages.

parameters   If not NULL, this is the command line parameters to pass to the loaded application. The forwarded character string might contain whitespaces and must be 0 terminated.

> If command line parameters are passed to a model that does not support this argument, the target must return `CADI_STATUS_ArgNotSupported`.

A.8.46 CADI::CADIExecUnLoadApplication()

This is used to unload symbol information of a specific image that was loaded previously.

```
virtual CADIReturn_t CADI::CADIExecUnloadApplication(const char *filename) =0;
```

where:

filename   is the same as was specified for `CADIExecLoadApplication()`.

A.8.47 CADI::CADIExecGetLoadedApplication()

This gets a list of image filenames that are currently loaded in the target.

```
virtual CADIReturn_t CADI::CADIExecGetLoadedApplications(uint32_t startIndex,
                                uint32_t desiredNumberOfApplications,
                                uint32_t *actualNumberOfApplicatiosReturnedOut,
                                char *filenamesOut,
```

```
                                            uint32_t filenameLength,
                                            char *parametersOut,
                                            uint32_t parametersLength) =0;
```

where:

startIndex   is the starting index in the list of filenames.

desiredNumberOfApplications

is the required number of applications (filename + parameters).

actualNumberOfApplicatiosReturnedOut

is the number of applications (filenames + parameters) that are valid in
`filenamesOut` and `parametersOut`.

filenamesOut

is a buffer of length [desiredNumberOfFilenames * filenamLength], the Nth
filename returned starts at offset N*filenameLength. The file name strings are zero
terminated.

filenameLength

is the maximum length of a single filename including terminating 0. Longer
filenames are truncated. All returned filenames must be 0 terminated. If one of the
returned filenames has the length `filenameLength-1` then `filenameLength` was too
short and must be redone. The target decides whether or not it can keep
information of more than one file.

parametersOut

is a buffer of length [desiredNumberOfApplications * parametersLength], the Nth
parameter returned starts at offset N*parametersLength. Each parameter string is
zero terminated. The target decides whether or not it can keep information for
more than one file.

parametersLength

is the maximum length of a single parameters string including terminating 0.
Longer parameters are truncated. All returned parameters must always be 0
terminated. If one of the returned parameters has the length `parametersLength-1`
then `parametersLength` was too short and must be redone. The target decides
whether or not it can keep information for more than one file.

A.8.48   CADI::CADIGetInstructionCount()

This method gets the current instruction count of the specific target that this debugger is
connected to.

```
virtual CADIReturn_t CADI::CADIGetInstructionCount(
        uint64_t & instructionCount) =0;
```

where:

instructionCount

is the returned instruction count.

A.8.49   CADI::CADIGetCycleCount()

This method gets the current cycle count.

```
virtual CADIReturn_t CADI::CADIGetCycleCount(uint64_t & cycleCount,
                                             bool systemCycles) =0;
```

where:

cycleCount   is the returned cycle count. This must be pre-initialized by the caller and assigned by the callee.

systemCycles

     if `true`, the method returns the system cycle count. If `false`, the method returns return the target specific cycle count.

———— **Note** ————

Not all targets support `cycleCount` or `systemCycles`. If not supported, the target returns either:
*   an approximation to the cycle count such as, for example, the instruction count
*   the error value `CADI_STATUS_CmdNotSupported`.

A.8.50   CADI::CADIBptGetList()

If the debugger attaches to a target that already has breakpoints set, this method enables the debugger to identify the breakpoints.

```
virtual CADIReturn_t CADI::CADIBptGetList(uint32_t startIndex,
                                          uint32_t desiredNumOfBpts,
                                          uint32_t * actualNumOfBpts,
                                          CADIBptDescription_t * breakpoints) =0;
```

where:

startIndex   is the index into the internal buffer of breakpoints held by the target.

desiredNumOfBpts

     is the required number of breakpoints.

actualNumOfBpts

     is the number of breakpoints that are actually returned in the buffer.

breakpoints   is an array of `CADIBptDescription_t` structures used to return the requested breakpoints (see *CADIBptDescription_t* on page B-25). The array must be allocated by the caller.

Vector catching

ARM Fast Models enable vector catching, using permanent breakpoints on special purpose registers. See Table A-1 for those common to AArch32 and AArch64, and Table A-2 on page A-44 for those unique to AArch64. `CADIBptGetList()` returns these breakpoints, if present, in addition to temporary ones.

Table A-1 Special purpose registers with permanent breakpoints, by processor and technology

Cortex-A and Cortex-R	TrustZone	TrustZone	Virtualization
RESET	-	-	-
UNDEFINED	NS_UNDEFINED	-	HYP_UNDEFINED
-	-	-	HYP_HYP

Table A-1 Special purpose registers with permanent breakpoints, by processor and technology (continued)

Cortex-A and Cortex-R	TrustZone	TrustZone	Virtualization
SVC	NS_SVC	SMC	HVC
PREFETCH_ABORT	NS_PREFETCH_ABORT	MON_PREFETCH_ABORT	HYP_PREFETCH_ABORT
DATA_ABORT	NS_DATA_ABORT	MON_DATA_ABORT	HYP_DATA_ABORT
IRQ	NS_IRQ	MON_IRQ	HYP_IRQ
FIQ	NS_FIQ	MON_FIQ	HYP_FIQ

Table A-2 Special purpose registers with permanent breakpoints unique to AArch64 processors[a]

Exception levels				Descriptor
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SP0_SYNC
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SP0_IRQ
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SP0_FIQ
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SP0_ABORT
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SPx_SYNC
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SPx_IRQ
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SPx_FIQ
S_EL1	NS_EL1	EL2	EL3	_CURRENT_SPx_ABORT
S_EL1	NS_EL1	EL2	EL3	_LOWER_64_SYNC
S_EL1	NS_EL1	EL2	EL3	_LOWER_64_IRQ
S_EL1	NS_EL1	EL2	EL3	_LOWER_64_FIQ
S_EL1	NS_EL1	EL2	EL3	_LOWER_64_ABORT
S_EL1	NS_EL1	EL2	EL3	_LOWER_32_SYNC
S_EL1	NS_EL1	EL2	EL3	_LOWER_32_IRQ
S_EL1	NS_EL1	EL2	EL3	_LOWER_32_FIQ
S_EL1	NS_EL1	EL2	EL3	_LOWER_32_ABORT

a. The names of these registers are made up of the exception level followed by the descriptor, for example
   S_EL1_CURRENT_SP0_SYNC.

A.8.51   CADI::CADIBptRead()

This method reads the breakpoint request information for a specific breakpoint ID. It can be used, for example, to retrieve the current `ignoreCount` of a specific breakpoint.

```
virtual CADIReturn_t CADIBptRead(CADIBptNumber_t breakpointId,
                                 CADIBptRequest_t *requestOut) =0;
```

where:

`breakpointId` is the ID of the breakpoint to be read.

`requestOut`   is the return buffer for a single breakpoint.

A.8.52　CADI::CADIBptSet()

This method sets a code breakpoint in the target.

```
virtual CADIReturn_t CADI::CADIBptSet(CADIBptRequest_t * request,
                                      CADIBptNumber_t * breakpoint) =0;
```

where:

request　　　is the requested breakpoint.

breakpoint　　is the resulting breakpoint (zero if the breakpoint was not set).

The CADIBptNumber_t is defined as uint32_t.

A.8.53　CADI::CADIBptClear()

This method removes a breakpoint from the target.

```
virtual CADIReturn_t CADI::CADIBptClear(CADIBptNumber_t breakpointId) =0;
```

where:

breakpointId　is the requested breakpoint.

──── **Note** ────

CADIBptClear() returns CADI_STATUS_IllegalArgument for permanent breakpoints.

A.8.54　CADI::CADIBptConfigure()

This method enables or disables a breakpoint in the target. This only applies if the target supports enabling and disabling of hardware breakpoints. Otherwise, this type of breakpoint management must be done on the host side.

```
virtual CADIReturn_t CADI::CADIBptConfigure(CADIBptNumber_t breakpointId,
                                            CADIBptConfigure_t configuration) =0;
```

where:

breakpointId　is the requested breakpoint.

configuration

　　　　　is the requested configuration.

A.9 The CADIDisassemblerCB class

This section describes the `CADIDisassemblerCB` class and its methods:

- *CADIDisassemblerCB class definition*
- *CADIDisassemblerCB::IFNAME()*
- *CADIDisassemblerCB::IFREVISION()*
- *CADIDisassemblerCB::ReceiveModeName()*
- *CADIDisassemblerCB::ReceiveSourceReference()* on page A-47
- *CADIDisassemblerCB::ReceiveDissassembly()* on page A-47.

A.9.1 CADIDisassemblerCB class definition

This callback class must be implemented by the disassembly front end. The definition is shown in Example A-12:

Example A-12 The CADIDisassemblerCB class

```
class CADI_WEXP CADIDisassemblerCB : public CAInterface
{
public:
    // Return the CAInterface name for this interface.
    static if_name_t IFNAME() { return "eslapi.CADIDisassemblerCB2"; }
    static if_rev_t IFREVISION() { return 0; }
    virtual void ReceiveModeName(uint32_t mode, const char *modename) =0;
    virtual void ReceiveSourceReference(const CADIAddr_t &addr, const char
            *sourceFile, uint32_t sourceLine) =0;
    virtual void ReceiveDisassembly(const CADIAddr_t &addr,
            const char *opcodes, const char *disassembly) =0;
};
```

A.9.2 CADIDisassemblerCB::IFNAME()

This callback returns the CAInterface name for this interface.

```
static if_name_t IFNAME() { return "eslapi.CADIDisassemblerCB2"; }
```

A.9.3 CADIDisassemblerCB::IFREVISION()

This callback specifies the current minor revision for this interface.

```
static if_rev_t IFREVISION() { return 0; }
```

A.9.4 CADIDisassemblerCB::ReceiveModeName()

This callback is triggered by `CADIDisassembler::GetModeNames()` and receives the mode name for the requested disassembler.

```
virtual void ReceiveModeName(uint32_t mode,
                             const char *modename) =0) =0;
```

where:

mode	is the required mode.
modename	returns the mode name string.

A.9.5 CADIDisassemblerCB::ReceiveSourceReference()

This callback is triggered by `CADIDisassembler::GetSourceReferenceForAddress()` and receives the source line and source file for the instruction at the requested address.

```
virtual ReceiveSourceReference(const CADIAddr_t &addr,
                               const char *sourceFile,
                               uint32_t sourceLine) =0;
```

where:

addr   is the requested address in the code.

sourceFile   is a reference to the source file for the requested address.

sourceline   is a reference to the source line for the requested address.

A.9.6 CADIDisassemblerCB::ReceiveDissassembly()

This callback is triggered by `CADIDisassembler::GetDisassembly()` and receives the requested disassembly.

```
virtual void ReceiveDisassembly(const CADIAddr_t &addr, const char *opcodes,
        const char *disassembly) =0;
```

where:

addr   is the requested address in the code.

opcodes   is the opcode text for the disassembled instruction.

disassembly   is the text for the disassembly.

A.10 The CADIDisassembler class

This section describes the CADIDisassembler class and its methods:

A.10.1 CADIDisassembler class definition

If a component supports disassembly, the Disassembly API can be used to display the disassembly during a simulation.

——— **Caution** ———

Obtaining a disassembler from the CADI interface by calling CADIGetDissambler() is deprecated. The function is retained to enable compatibility with CADI 1.1.

New code must use the ObtainInterface() call for both disassembler and profiling support.

——— **Note** ———

A program memory space must exist to use the disassembly feature.

Example A-13 CADIDisassembler class

```
class CADIDisassembler  : public CAInterface
{
public:
    static if_name_t IFNAME() { return "eslapi.CADIDisassembler2"; }
    static if_rev_t IFREVISION() { return 0; }

    // Two types: distinguish standard and history type
    virtual CADIDisassemblerType GetType() const =0;

    // Support for multiple modes (e.g. 32bit versus 16bit mode)
    virtual uint32_t GetModeCount() const =0;
    virtual void GetModeNames(CADIDisassemblerCB *callback) =0;
    virtual uint32_t GetCurrentMode() =0;
    virtual CADIDisassemblerStatus GetSourceReferenceForAddress(
        CADIDisassemblerCB *callback, const CADIAddr_t &address) =0;
    virtual CADIDisassemblerStatus GetAddressForSourceReference(
        const char *sourceFile, uint32_t sourceLine, CADIAddr_t &address) =0

    //    function for standard type disassembly
    virtual CADIDisassemblerStatus GetDisassembly(CADIDisassemblerCB *callback,
        const CADIAddr_t &address, CADIAddr_t &nextAddr, const uint32_t mode,
        uint32_t desiredCount = 1) =0;

    // Query if an instruction is a call instruction
```

```
virtual CADIDisassemblerStatus GetInstructionType(const CADIAddr_t &address,
    CADIDisassemblerInstructionType &insn_type) =0;

// A default minimum implementation, to provide backwards-compatibility.
// This implementation assumes that there will be no other interfaces
// implemented on the component providing CADIDisassembler.
virtual CAInterface * ObtainInterface(if_name_t ifName, if_rev_t minRev,
    if_rev_t *   actualRev)
    {
        if((strcmp(ifName,IFNAME()) == 0) && (minRev <= IFREVISION()))
        {
            if (actualRev) // make sure this is not a NULL pointer
            {
                *actualRev = IFREVISION();
            }
          return this;
        }
        if((strcmp(ifName, CAInterface::IFNAME()) == 0) &&
            minRev <= CAInterface::IFREVISION())
        {
            if (actualRev != NULL)
            {
                *actualRev = CAInterface::IFREVISION();
            }
            return this;
        }
        return NULL;
    }
};
```

A.10.2   CADIDisassembler::GetType()

The return value indicates whether the type is standard, source level, or interpretive.

```
virtual CADIDisassemblerType CADIDisassembler::GetType() const =0;
```

The types are defined in the enumeration:

```
enum CADIDisassemblerType
{
   CADI_DISASSEMBLER_TYPE_STANDARD,  //disassembly supporting a PC and lookahead
   CADI_DISASSEMBLER_TYPE_SOURCELEVEL=2, //source level assembly / C
   CADI_DISASSEMBLER_TYPE_INTERPRETER   // interpreter window (for scripts)
};
```

A.10.3   CADIDisassembler::GetModeCount()

The return value from this function indicates support for multiple modes such as, for example, 32bit or 16bit mode. Valid modes start at 1. Mode 0 indicates no modes or *don't care*.

```
virtual uint32_t CADIDisassembler::GetModeCount() =0;
```

A.10.4   CADIDisassembler::GetModeNames()

This function returns the name of all modes. The callback CADIDisassemblerCB::ReceiveModeName() is triggered once for every mode.

```
virtual std::string CADIDisassembler::GetModeNames(
                                    CADIDisassemblerCB *callback) =0;
```

A.10.5 CADIDisassembler::GetCurrentMode()

The return value indicates the current execution mode. If modes are not supported by this target, the return value is 0. If modes are supported, the return value is a number between 1 and the value returned by `GetModeCount()`.

```
virtual uint32_t CADIDisassembler::GetCurrentMode() =0;
```

A.10.6 CADIDisassembler::GetSourceReferenceForAddress()

The is used to obtain source-level information. It triggers the `CADIDisassemblerCB::ReceiveSourceReference()` callback.

```
virtual CADIDisassemblerStatus CADIDisassembler::GetSourceReferenceForAddress(
                                            CADIDisassemblerCB *callback,
                                            const CADIAddr_t &address) =0;
```

where:

callback   is the callback object to receive the source-level information.

address   is the address the source-level information is requested for.

A.10.7 CADIDisassembler::GetAddressForSourceReference()

This method is used to obtain the first address for a specified source line in a specified file.

```
virtual CADIDisassemblerStatus CADIDisassembler::GetAddressForSourceReference(
                                            const char *sourceFile,
                                            uint32_t sourceLine,
                                            CADIAddr_t &address) =0;
```

where:

sourceLine   is the requested source line number.

sourceFile   is a null terminated C string containing the source file name.

address   is set to the address corresponding to the source line and file.

A.10.8 CADIDisassembler::GetDisassembly()

Function enables standard type disassembly. Each disassembled instruction triggers the `CADIDisassembler::ReceiveDisassembly()` callback.

```
virtual CADIDisassemblerStatus CADIDisassembler::GetDisassembly(
                                            CADIDisassemblerCB *callback,
                                            const CADIAddr_t &address,
                                            CADIAddr_t &nextAddr,
                                            const uint32_t mode,
                                            uint32_t desiredCount = 1) =0;
```

where:

callback   is the callback object to receive the disassembly.

address   passes the address of the instruction to disassemble and to return the address of the next valid instruction. Mandatory if the return value is `CADI_DISASSEMBLER_STATUS_NO_INSTRUCTION` or `CADI_DISASSEMBLER_STATUS_ILLEGAL_ADDRESS`.

nextAddr   returns the address of the next instruction. This must be used if the return value is `CADI_DISASSEMBLER_STATUS_NO_INSTRUCTION` or `CADI_DISASSEMBLER_STATUS_ILLEGAL_ADDRESS`.

nextAddr must be a hint to the next address that might result in successful disassembly.

mode   contains the execution mode. If 0, use the current execution mode.

desiredCount can be used to disassemble a sequence of instructions. Up to *desiredCount* calls are made to CADIDisassemblerCB::ReceiveDisassembly().

The first instruction is the instruction pointed to by *address*. The sequence of disassembled instructions stops if an error such as, for example, no instruction or illegal address, occurs while attempting to disassemble an instruction

return value is the status. The possible values are defined by the CADIDisassemblerStatus enumeration. See *CADIDisassemblerStatus* on page B-31.

A.10.9 CADIDisassembler::GetInstructionType()

This method determines whether the instruction is a call instruction.

```
virtual CADIDisassemblerStatus GetInstructionType(const CADIAddr_t &address,
                            CADIDisassemblerInstructionType &insn_type) =0;
```

where:

address   is used to pass the address of the instruction to check.

insn_type   is true if the instruction is a call instruction
(CADI_DISASSEMBLER_INSTRUCTION_TYPE_CALL).

A.10.10 CADIDisassembler::ObtainInterface()

This is a default minimum implementation. This implementation assumes that there are no other interfaces implemented on the component that provide CADIDisassembler.

```
virtual CAInterface * ObtainInterface(if_name_t ifName, if_rev_t minRev,
                            if_rev_t * actualRev)
```

See CADIDisassembler.h for implementation details.

A.11   The CADIProfilingCallbacks class

This section describes the CADIProfilingCallbacks class and its methods:

- *CADIProfilingCallbacks class definition*
- *CADIProfilingCallbacks::profileResourceAccess()*
- *CADIProfilingCallbacks::profileRegisterHazard()*.

A.11.1   CADIProfilingCallbacks class definition

The definition of CADIProfilingCallbacks is:

Example A-14 The CADIProfilingCallbacks class

```
class CADI_WEXP CADIProfilingCallbacks :
        public CAInterface
{
public:
 static if_name_t IFNAME() { return "eslapi.CADIProfilingCallbacks2"; }
 static if_rev_t IFREVISION() { return 0; }
 virtual void profileResourceAccess(const char * name,
                      CADIProfileResourceAccessType_t accessType) = 0;
 virtual void profileRegisterHazard(CADIProfileHazardDescription_t * desc) = 0;
};
```

A.11.2   CADIProfilingCallbacks::profileResourceAccess()

Profiles a resource access that has been registered by CADIRegisterProfileResourceAccess().

```
virtual CADIReturn_t CADIProfilingCallback::profileResourceAccess(
                              const char * name,
                              CADIProfileResourceAccessType_t accessType) =0;
```

where:

name   is the name of the resource.

accessType   specifies the read/write access as defined in *CADIProfileResourceAccessType_t*
              on page B-37.

A.11.3   CADIProfilingCallbacks::profileRegisterHazard()

Report that a hazard has occurred of type CADIProfileHazardDescription_t.

```
virtual CADIReturn_t CADIProfilingCallback::profileRegisterHazard(
        CADIProfileHazardDescription_t desc) =0;
```

where:

desc   is of type CADIProfileHazardDescription_t (see
              CADIProfileHazardDescription_t on page B-37).

A.12 The CADIProfiling class

This section describes the CADIProfiling class and its methods. It enables you to record and monitor profile information related to the debugging session. The class has these methods:

A.12.1 CADIProfiling class definition

The CADIProfiling class definition is shown in Example A-15:

Example A-15 The CADIProfiling class

```
class CADI_WEXP CADIProfiling : public CAInterface
{
public:
   static if_rev_t IFREVISION() { return 0; }
   virtual CADIReturn_t CADIProfileSetup (CADIProfileType_t type,
                        uint32_t regionCount, CADIProfileRegion_t *region) = 0;
   virtual CADIReturn_t CADIProfileControl (CADIProfileControl_t control) = 0;
   virtual CADIReturn_t CADIProfileTraceControl (CADITraceBufferControl_t bufferArg,
                        CADITraceControl_t control, CADITraceOverlayControl_t overlay) = 0;
   virtual CADIReturn_t CADIProfileGetExecution (CADIProfileResultType_t *type, uint32_t regIndex,
                        uint32_t regionSlots, uint32_t *regionCount,
                        CADIProfileResults_t *region) = 0;
   virtual CADIReturn_t CADIProfileGetMemory (CADIProfileResultType_t *type,  uint32_t regIndex,
                        uint32_t regionSlots, uint32_t *regionCount,
                        CADIProfileResults_t *region) = 0;
   virtual CADIReturn_t CADIProfileGetTrace (uint32_t blockIndex, uint32_t blockSlots,
                        uint32_t *blockCount, CADITraceBlock_t *block) = 0;
   virtual CADIReturn_t CADIProfileGetRegAccesses(uint32_t startRegID, uint32_t numberOfRegs,
```

```
                        CADIRegProfileResults_t *reg, uint32_t &actualNumberOfRegs) = 0;
  virtual CADIReturn_t CADIProfileSetRegAccesses(uint32_t startRegID, uint32_t numberOfRegs,
                        CADIRegProfileResults_t *reg, uint32_t &actualNumberOfRegs) = 0;
  virtual CADIReturn_t CADIProfileGetMemAccesses(CADIAddrComplete_t startAddress,
                        uint32_t numberOfUnits, CADIMemProfileResults_t *mem,
                        uint32_t &actualNumberOfUnits) = 0;
  virtual CADIReturn_t CADIProfileSetMemAccesses(CADIAddrComplete_t startAddress,
                        uint32_t numberOfUnits, CADIMemProfileResults_t *mem,
                        uint32_t &actualNumberOfUnits) = 0;
  virtual CADIReturn_t CADIProfileGetAddrExecutionFrequency(uint64_t startAddr, uint32_t numberOfAddr,
                        uint64_t *freq, uint32_t &actualNumberOfAddr) = 0;
  virtual CADIReturn_t CADIProfileSetAddrExecutionFrequency(uint64_t startAddr, uint32_t numberOfAddr,
                        uint64_t *freq, uint32_t &actualNumberOfAddr) = 0;
  virtual CADIReturn_t CADIGetNumberOfInstructions(uint32_t *num_instructions) = 0;
  virtual CADIReturn_t CADIProfileInitInstructionResultArray(uint32_t numberOfInstructions,
                        CADIInstructionProfileResults_t *instructions,
                        uint32_t &actualNumberOfInstructions) = 0;
  virtual CADIReturn_t CADIProfileGetInstructionExecutionFrequency(uint32_t  numberOfInstructions,
                        CADIInstructionProfileResults_t *instructions,
                        uint32_t  &actualNumberOfInstructions ) = 0;
  virtual CADIReturn_t CADIProfileSetInstructionExecutionFrequency(uint32_t  numberOfInstructions,
                        CADIInstructionProfileResults_t *instructions,
                        uint32_t  &actualNumberOfInstructions ) = 0;
  virtual CADIReturn_t CADIRegisterProfileResourceAccess(const char * name,
                        CADIProfileResourceAccessType_t accessType) = 0;
  virtual CADIReturn_t CADIUnregisterProfileResourceAccess(const char * name) = 0;
  virtual CADIReturn_t CADIProfileRegisterCallBack(CADIProfilingCallbacks *callbackObject) = 0;
  virtual CADIReturn_t CADIProfileUnregisterCallBack(CADIProfilingCallbacks *callbackObject) = 0;
};
```

A.12.2   CADIProfiling::CADIProfileSetup()

This informs the target of the memory regions that are to be profiled. This function must be called only once before any number of calls to either of the following:

- CADIProfileControl(CADI_PROF_CNTL_Start)

- CADIProfileControl(CADI_PROF_CNTL_Stop).

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetup(CADIProfileType_t type,
            uint32_t regionCount, CADIProfileRegion_t * region) =0;
```

where:

type   is the type of profiling, execution addresses or data access, to which these regions apply. It is one of the values defined in *CADIProfileType_t* on page B-35:

- CADI_PROF_TYPE_Execution

- CADI_PROF_TYPE_Memory is used with CADIProfileGetMemory()

- CADI_PROF_TYPE_Trace is used with CADIProfileGetTrace().

regionCount   is the number of regions.

region   contains the description of the memory areas being added (see *CADIProfileRegion_t* on page B-34). The caller allocates the required memory for this array.

The return value must be CADI_STATUS_IllegalArgument if any of the following are true:

- any region spans unpopulated memory

- any region spans illegal memory

- any region overlaps another region

- the address space of a region is not consistent with the profiling type.

A.12.3 CADIProfiling::CADIProfileControl()

This starts, stops, or resets profiling by passing a member of the `CADIProfileControl_t` enumeration.

```
virtual CADIReturn_t CADIProfiling::CADIProfileControl(
                          CADIProfileControl_t control) =0;
```

where:

control   defines profiling behavior (see *CADIProfileControl_t* on page B-35).

─── **Note** ───

Starting profiling resets any saved information. Stopping profiling does not reset recorded information.

─────────────

A.12.4 CADIProfiling::CADIProfileTraceControl()

This starts, stops, and resets recording the execution trace.

```
virtual CADIReturn_t CADIProfiling::CADIProfileTraceControl(
                          CADITraceBufferControl_t bufferArg,
                          CADITraceControl_t control,
                          CADITraceOverlayControl_t overlay) =0;
```

where:

bufferArg   is the action to take place when the buffer is full, that is, either wrap or stop. See *CADITraceBufferControl_t* on page B-38.

control   defines the tracing behavior and is one of the values defined in *CADITraceControl_t* on page B-38:

- CADI_TRACE_CNTL_StartContinuous
- CADI_TRACE_CNTL_StartDiscontinuity
- CADI_TRACE_CNTL_Stop.

overlay   selects overlay and is one of the values defined in *CADITraceOverlayControl_t* on page B-39:

- If CADI_TRACE_OVERLAY_Memory, overlay events must be included in the trace output at the expense of not being able to see inside the trace manager.

- If CADI_TRACE_OVERLAY_Manager, the trace data must include the overlay manager code at the expense of not knowing the details about the memory regions that are overlaid.

A.12.5 CADIProfiling::CADIProfileGetExecution()

This gets the results of a profiling session for executable code.

If called before profiling is stopped or before a legal set of regions has been established, this call must return CADI_STATUS_GeneralError.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetExecution(
                          CADIProfileResultType_t * type,
                          uint32_t regIndex, uint32_t regionSlots,
                          uint32_t * regionCount,
                          CADIProfileResults_t * region) =0;
```

where:

type indicates whether percentage statistics or an absolute count is being returned. See *CADIProfileResultType_t* on page B-34.

regIndex is the index into the internal buffer held by the target.

regionSlots is the number of spaces requested to be filled. The target shall not fill more than this number of elements in the region array.

regionCount is the actual number of regions set up by CADIProfileSetup plus one. The additional count indicates the other category.

region corresponds to the regions set up by CADIProfileSetup. The array is allocated, and deallocated if applicable, by the caller and filled by the target. See *CADIProfileResults_t* on page B-34.

A.12.6 CADIProfiling::CADIProfileGetMemory()

This gets the results of a profiling session for memory accesses. If called before profiling is stopped or before a legal set of profiling regions has been established, the return value must be CADI_STATUS_GeneralError.

CADIProfileGetMemory() is similar to CADIProfileGetExecution(). It enables future versions to separately modify the call signatures of the two functions.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetMemory(
                                  CADIProfileResultType_t * type,
                                  uint32_t regIndex,
                                  uint32_t regionSlots,
                                  uint32_t * regionCount,
                                  CADIProfileResults_t * region) =0;
```

where:

type tells the caller whether percentage statistics or an absolute count is being returned. See *CADIProfileResultType_t* on page B-34.

regIndex is the index into the internal buffer held by the target.

regionSlots is the number of spaces requested to be filled. The target shall not fill more than this number of elements in the region array.

regionCount is the actual number of regions set up by CADIProfileSetup plus one. The additional count indicates the other category.

region corresponds to the regions set up by CADIProfileSetup. The array is allocated, and deallocated if applicable, by the caller and filled by the target. See *CADIProfileResults_t* on page B-34.

A.12.7 CADIProfiling::CADIProfileGetTrace()

This gets the results of a trace session. The block parameter contains the PC values that have been executed by the target.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetTrace(uint32_t blockIndex,
                                  uint32_t blockSlots,
                                  uint32_t * blockCount,
                                  CADITraceBlock_t * block) =0;
```

where:

blockIndex   is the start index of the trace block.

blockSlots   is the number of spaces available to fill. The target must not fill more than this number of elements in the block array.

blockCount   is the number of samples being returned.

block   is the list of executed addresses and overlay events in time sequential order. The blocks in the array must be sorted by time executed and block[0] must contain the most recently executed address or event. If multiple program memory spaces exist, and execution uses multiple spaces during execution, separate blocks must exist for each memory space. The block array is allocated, and deallocated if applicable, by the caller and filled in by the target. See *CADITraceBlock_t* on page B-39.

A.12.8 CADIProfiling::CADIProfileGetRegAccesses()

Reads the number of read/write accesses for *numberOfRegs* registers, starting with register index *startReg*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetRegAccesses(
                                uint32_t startRegID,
                                uint32_t numberOfRegs,
                                CADIRegProfileResults_t * reg,
                                uint32_t & actualNumberOfRegs) =0;
```

where:

startRegID   is the index of the first profiled register in the internal list of profiled registers held by the target.

NumberOfRegs is the number of registers the profiling data is requested for.

reg   on return, this contains the profiling results. See *CADIProfileResults_t* on page B-34.

> **Note**
>
> *reg* must point to an array of objects of type CADIResourceProfileResults_t with size *numberOfRegs*.

actualNumberOfRegs

on return, this contains the number of registers the profiling data was actually read for.

A.12.9 CADIProfiling::CADIProfileSetRegAccesses()

Writes the number of read/write accesses to the profiling resources for *numberOfRegs* registers according to values saved in *reg*, starting with register index *startReg*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetRegAccesses(
                                uint32_t startRegID,
                                uint32_t numberOfRegs,
                                CADIRegProfileResults_t * reg,
                                uint32_t & actualNumberOfRegs) =0;
```

where:

startRegID   is the index of the first profiled register in the internal list of profiled registers held
              by the target.

NumberOfRegs is the number of registers the profiling data is set for.

reg   contains the results to use to set the profiling resources. See *CADIProfileResults_t*
              on page B-34.

—— **Note** ——

reg must point to an array of objects of type CADIResourceProfileResults_t with
size *numberOfRegs*.

actualNumberOfRegs

contains the number of actually updated registers.

A.12.10 CADIProfiling::CADIProfileGetMemAccesses()

Reads the number of read/write accesses for *numberOfRegs* memory units.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetMemAccesses(
                                    CADIAddrComplete_t startAddress,
                                    uint32_t numberOfUnits,
                                    CADIMemProfileResults_t * mem,
                                    uint32_t & actualNumberOfUnits) =0;
```

where:

startAddress is the start address for the selected memory units. See *CADIAddrComplete_t* on
              page B-20.

numberOfUnits

is the number of selected memory units.

mem   contains the results on return. See *CADIMemProfileResults_t* on page B-36.

—— **Note** ——

mem must point to an array of objects of type CADIResourceProfileResults_t with
size *numberOfUnits*.

actualNumberOfUnits

contains the actual number of memory units for which data was collected.

A.12.11 CADIProfiling::CADIProfileSetMemAccesses()

Writes the number of read/write accesses to the profiling resources for *numberOfUnits* memory
units according to values saved in *mem*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetMemAccesses(
                                    CADIAddrComplete_t startAddress,
                                    uint32_t numberOfUnits,
                                    CADIMemProfileResults_t * mem,
                                    uint32_t & actualNumberOfUnits) =0;
```

where:

startAddress is the starting address for the memory units. See *CADIAddrComplete_t* on
              page B-20.

NumberOfUnits

> is the number of memory units.

mem   contains the values to use for the update of the profiling resources. See *CADIMemProfileResults_t* on page B-36.

> ——— **Note** ———
>
> *mem* must point to an array of objects of type CADIMemProfileResults_t with size *numberOfUnits*.

actualNumberOfUnits

> contains the number of memory units for which data was actually updated.

A.12.12 CADIProfiling::CADIProfileGetAddrExecutionFrequency()

Reads the execution frequency for *numberOfAddr* disassembly addresses.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetAddrExecutionFrequency(
        uint64_t startAddr, uint32_t numberOfAddr, uint64_t * freq,
        uint32_t & actualNumberOfAddr) =0;
```

where:

startAddr   is the start address for the requested disassembly addresses.

numberOfAddr

> is the number of requested disassembly addresses.

freq   contains the results on return.

> ——— **Note** ———
>
> *freq* must point to an array of uint64_t with size *numberOfAddr*.

actualNumberOfAddr

> contains the actual number of disassembly addresses for which the frequency was read.

A.12.13 CADIProfiling::CADIProfileSetAddrExecutionFrequency()

Writes the execution frequency for numberOfAddr disassembly addresses to the profiling resources according to values saved in freq.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetAddrExecutionFrequency(
                              uint64_t startAddr,
                              uint32_t numberOfAddr,
                              uint64_t * freq,
                              uint32_t & actualNumberOfAddr) =0;
```

where:

startAddr   is the start address for the requested disassembly addresses.

numberOfAddr

> is the number of requested disassembly addresses.

freq   contains the values to use to update the disassembly addresses.

───── **Note** ─────

freq must point to an array of `uint64_t` with size *numberOfAddr*.

actualNumberOfAddr

> contains the actual number of disassembly addresses for which the profiling resources were updated.

A.12.14 CADIProfiling::CADIGetNumberOfInstructions()

Returns number of instructions of the target.

```
virtual uint32_t CADIProfiling::CADIGetNumberOfInstructions() =0;
```

A.12.15 CADIProfiling::CADIProfileInitInstructionResultArray()

This method prepares instruction profiling according to the given array *instructions* by setting *FID*, *name*, and *pathToInstructionInLISASource*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileInitInstructionResultArray(
                    uint32_t numberOfInstructions,
                    CADIInstructionProfileResults_t * instructions,
                    uint32_t & actualNumberOfInstructions) =0;
```

where:

numberOfInstructions

> is the required number of array entries to be prepared.

instructions is an array that contains the values to use for preparing profiling. See *CADIInstructionProfileResults_t* on page B-36.

actualNumberOfInstructions

> is the number of array entries actually prepared.

A.12.16 CADIProfiling::CADIProfileGetInstructionExecutionFrequency()

Reads the execution counts for *numberOfInstructions* instructions by setting the appropriate *executionCount* entry in array *instructions*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileGetInstructionExecutionFrequency(
                    uint32_t numberOfInstructions,
                    CADIInstructionProfileResults_t * instructions,
                    uint32_t & actualNumberOfInstructions) =0;
```

where:

numberOfInstructions

> is the required number of instructions to read to the profiling resources.

instructions is an array to contain the results. See *CADIInstructionProfileResults_t* on page B-36.

actualNumberOfInstructions

> is the number of instructions actually read.

A.12.17 CADIProfiling::CADIProfileSetInstructionExecutionFrequency()

Writes the execution counts for `numberOfInstructions` instructions according to values in *instructions*.

```
virtual CADIReturn_t CADIProfiling::CADIProfileSetInstructionExecutionFrequency(
                      uint32_t numberOfInstructions,
                      CADIInstructionProfileResults_t * instructions,
                      uint32_t & actualNumberOfInstructions) =0;
```

where:

numberOfInstructions

is the required number of array entries to write to the target.

instructions   contains the values to write to the target. See *CADIInstructionProfileResults_t on page B-36*.

actualNumberOfInstructions

is the number of array entries actually written to the target.

A.12.18 CADIProfiling::CADIRegisterProfileResourceAccess()

Registers a resource access callback.

```
virtual CADIReturn_t CADIProfiling::CADIProfileRegisterResourceAccess(
                      const char * name,
                      CADIProfileResourceAccessType_t accessType) =0;
```

where:

name   is a resource.

accessType   is one of the values defined in *CADIProfileResourceAccessType_t on page B-37*:
   • CADI_PROF_ACCESS_READ
   • CADI_PROF_ACCESS_WRITE
   • CADI_PROF_ACCESS_READ_OR_WRITE.

A.12.19 CADIProfiling::CADIUnregisterProfileResourceAccess()

Unregisters the resource access callback.

```
virtual CADIReturn_t CADIProfiling::CADIProfileUnregisterResourceAccess(
                      const char * name) =0;
```

A.12.20 CADIProfiling::CADIProfileRegisterCallBack()

Registers a profiling callback to the target.

```
virtual CADIReturn_t CADIProfiling::CADIProfileRegisterCallBack(
                      CADIProfilingCallbacks * callBackObject) =0;
```

where:

callBackObject

is the callback. See *The CADIProfilingCallbacks class on page A-52.*.

A.12.21 CADIProfiling::CADIProfileUnregisterCallBack()

Unregisters a profiling callback from the target.

```
virtual CADIReturn_t CADIProfiling::CADIProfileUnregisterCallBack(
                      CADIProfilingCallbacks *callbackObject) =0;
```

where:

callBackObject

is the callback. See *The CADIProfilingCallbacks class* on page A-52..

Appendix B
Data Structures Used by the CADI Interface

This chapter describes data structures used by CADI. It contains the following sections:

—— **Note** ——

For the full list of data structures and types, see the CADI header files.

B.1 Factory simulation startup and configuration

This section describes data types associated with CADI configuration.

B.1.1 CADIReturn_t

This is the result returned by most calls and it is a general indication of the status of the call. When an error is detected, the debugger can call `CADIXfaceGetError()` to retrieve an error message in text form.

Example B-1 CADIReturn_t

```
enum CADIReturn_t
{
   CADI_STATUS_OK,                   // The call was successful.
   CADI_STATUS_GeneralError,         // This indicates an error that isn't sufficiently explained by
                                     // one of the other error status values.
   CADI_STATUS_UnknownCommand,       // The command is not recognized.
   CADI_STATUS_IllegalArgument,      // An argument value is illegal.
   CADI_STATUS_CmdNotSupported,      // The command is recognized but not supported.
   CADI_STATUS_ArgNotSupported,      // An argument to the command is recognized but not supported.
                                     // For example, the target does not support a
                                     // particular type of complex breakpoint.
   CADI_STATUS_InsufficientResources, // Not enough memory or other resources
                                     // exist to fulfill the command.
   CADI_STATUS_TargetNotResponding,  // A timeout has occurred across the CADI interface
                                     // - the target did not respond to the command.
   CADI_STATUS_TargetBusy,           // The target received a request, but is unable to process the
                                     // command. The caller can try this call again after some time.
   CADI_STATUS_BufferSize,           // Buffer too small (for char* types)
   CADI_STATUS_SecurityViolation,    // Request has not been fulfilled due to
                                     // a security violation
   CADI_STATUS_PermissionDenied,     // Request has not been fulfilled since
                                     // the permission was denied
   CADI_STATUS_ENUM_MAX = 0xFFFFFFFF // Max enum value.
};
```

B.1.2 CADIFactoryErrorCode_t

The `CADIFactoryErrorCode_t` type specifies the values for the different error conditions.

Example B-2 CADIFactoryErrorCode_t

```
enum CADIFactoryErrorCode_t
{
        CADIFACT_ERROR_OK,            // no error at all, message is empty
        // license checking
        CADIFACT_ERROR_LICENSE_FOUND_BUT_EXPIRED,
        CADIFACT_ERROR_LICENSE_NOT_FOUND,
        CADIFACT_ERROR_LICENSE_COUNT_EXCEEDED,
        CADIFACT_ERROR_CANNOT_CONTACT_LICENSE_SERVER,
        CADIFACT_ERROR_WARNING_LICENSE_WILL_EXPIRE_SOON,   // always warning = true
        CADIFACT_ERROR_GENERAL_LICENSE_ERROR,     // for all other license errors
        // info: the parameter which caused this error is indicated in erroneousParameterId
        CADIFACT_ERROR_PARAMETER_TYPE_MISMATCH,  // dataType != dataType
        CADIFACT_ERROR_PARAMETER_VALUE_OUT_OF_RANGE,
        CADIFACT_ERROR_PARAMETER_VALUE_INVALID,  // not out of range but still invalid
```

```
        CADIFACT_ERROR_UNKNOWN_PARAMETER_ID,
        CADIFACT_ERROR_GENERAL_PARAMETER_ERROR, // for all other errors concerning a specific parameter
        CADIFACT_ERROR_GENERAL_ERROR,           // other, for everything else which prevented the CADI
                                                // interface from being created
        CADIFACT_ERROR_GENERAL_WARNING,     // always warning = true, for everything else which still
                                            // allowed the CADI interface to be created
        CADIFACT_ERROR_MAX = 0xFFFFFFFF
};
```

B.1.3 CADIFactorySeverityCode_t

The severity code is based on the to the error codes in `CADIFactoryErrorCode_t` and enables easy detection of errors and warnings.

Example B-3 CADIFactorySeverityCode_t

```
enum CADIFactorySeverityCode_t
{
        CADIFACT_SEVERITY_OK,           // no error at all, model created
        CADIFACT_SEVERITY_WARNING,      // only a warning, model still created
        CADIFACT_SEVERITY_ERROR,        // error, model not created
        CADIFACT_SEVERITY_MAX = 0xFFFFFFFF
};
```

B.1.4 CADISimulationInfo_t

This structure contains details about a simulation.

Example B-4 CADISimulationInfo_t

```
struct CADISimulationInfo_t
{
    public: // methods
        CADISimulationInfo_t(uint32_t id = 0,
                                const char *name_par = "",
                                const char *description_par = "") :
        id(id)
        {
            AssignString(name, name_par, CADI_NAME_SIZE);
            AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
        }
    public: // data
        uint32_t   id;
        char name[CADI_NAME_SIZE];
        char description[CADI_DESCRIPTION_SIZE];
};
```

Where:

id   is used for identification.

name   is the simulation name.

description   is the simulation description.

B.1.5   CADIParameterInfo_t

CADIParameterInfo_t and CADIParameterValue_t structures are used to configure component parameters.

```
struct CADIParameterInfo_t
{
public: // methods
    CADIParameterInfo_t(uint32_t id=0,
            const char *name_par="",
            CADIValueDataType_t dataType=CADI_PARAM_INVALID,
            const char *description_par = "",
            uint32_t isRunTime = 0,
            int64_t minValue = 0,
            int64_t maxValue = 0,
            int64_t defaultValue = 0,
            const char *defaultString_par = "") :
        id(id), dataType(dataType),
        isRunTime(isRunTime),
        minValue(minValue),
        maxValue(maxValue),
        defaultValue(defaultValue)
        {
          AssignString(name, name_par, CADI_NAME_SIZE);
          AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
          AssignString(defaultString, defaultString_par,
          CADI_DESCRIPTION_SIZE);
        }
public: // data
      uint32_t   id;
      char   name[CADI_NAME_SIZE];
      CADIValueDataType_t dataType;
      char   description[CADI_DESCRIPTION_SIZE];
      uint32_t   isRunTime;
      int64_t   maxValue;
      int64_t   defaultValue;
      char   defaultString[CADI_DESCRIPTION_SIZE];
};
```

Where:

id	is used for identification.
name	is the name of the parameter.
dataType	is the data type for interpretation purposes of the debugger.
description	is the parameter description.
isRunTime	if 0, the parameter is instantiation-time only. If 1, the parameter can be changed at run-time.
minValue	is the minimum admissible value.
maxValue	is the maximum admissible value.
defaultValue	is the default value if parameter is type bool/int.
defaultString	is the default string if parameter is type CADI_PARAM_STRING.

B.1.6 CADIParameterValue_t

CADIParameterInfo_t and CADIParameterValue_t structures are used to configure component parameters.

Example B-6 CADIParameterValue_t

```
struct CADIParameterValue_t
{
public: // methods
    CADIParameterValue_t(uint32_t parameterID = static_cast<uint32_t>(-1),
                         CADIValueDataType_t dataType=CADI_PARAM_INVALID,
                         int64_t intValue = 0,
                         const char *stringValue_par="") :
            parameterID(parameterID),
            dataType(dataType),
            intValue(intValue)
        {
            AssignString(stringValue, stringValue_par, CADI_DESCRIPTION_SIZE);
        }
public: // data
        uint32_t    parameterID;
        CADIValueDataType_t dataType;
        int64_t    intValue;
        char  stringValue[CADI_DESCRIPTION_SIZE];
};
```

Where:

parameterID refers to the id of respective CADIParameterInfo_t.

dataType   is the data type for interpretation by the debugger.

description is the parameter description.

intValue   is the BOOL (0 = false, 1 = true).

maxValue   is the maximum admissible value.

stringValue is the string value if the type is string.

B.1.7 CADITargetFeatures_t structure

The CADITargetFeatures_t structure is used by the CADIXfaceGetFeatures() call. The structure has the constructor as listed in Example B-7:

Example B-7 CADITargetFeatures_t structure

```
CADITargetFeatures_t(const char *targetName_par = "",
                     const char *targetVersion_par = "",
                     uint32_t nrBreakpointsAvailable_par = 0,
                     uint8_t fOverlaySupportAvailable_par = 0,
                     uint8_t fProfilingAvailable_par = 0,
                     uint32_t nrResetLevels_par = 0,
                     uint32_t nrExecModes_par = 0,
                     uint32_t nrExceptions_par = 0,
                     uint32_t nrMemSpaces_par = 0,
                     uint32_t nrRegisterGroups_par = 0,
                     uint32_t nrPipeStages_par = 0,
                     uint32_t nPCRegNum_par = CADI_INVALID_REGISTER_ID,
                     uint16_t handledBreakpoints_par = 0,
                     uint32_t nrOfHWThreads_par = 0,
```

```
                    uint32_t nExtendedTargetFeaturesRegNum_par = CADI_INVALID_REGISTER_ID,
                    char const* canonicalRegisterDescription_par = "",
                    char const* canonicalMemoryDescription_par = "",
                    uint8_t canCompleteMultipleInstructionsPerCycle_par = 0
        ) :
    nrBreakpointsAvailable(nrBreakpointsAvailable_par),
    fOverlaySupportAvailable(fOverlaySupportAvailable_par),
    fProfilingAvailable(fProfilingAvailable_par),
    nrResetLevels(nrResetLevels_par),
    nrExecModes(nrExecModes_par),
    nrExceptions(nrExceptions_par),
    nrMemSpaces(nrMemSpaces_par),
    nrRegisterGroups(nrRegisterGroups_par),
    nrPipeStages(nrPipeStages_par),
    nPCRegNum(nPCRegNum_par),
    handledBreakpoints(handledBreakpoints_par),
    nrOfHWThreads(nrOfHWThreads_par),
    nExtendedTargetFeaturesRegNumValid(nExtendedTargetFeaturesRegNum_par !=CADI_INVALID_REGISTER_ID),
    nExtendedTargetFeaturesRegNum(nExtendedTargetFeaturesRegNum_par),
    canCompleteMultipleInstructionsPerCycle(canCompleteMultipleInstructionsPerCycle_par)
    {
        AssignString(targetName, targetName_par, sizeof(targetName));
        AssignString(targetVersion, targetVersion_par, sizeof(targetVersion));
        AssignString(canonicalRegisterDescription, canonicalRegisterDescription_par,
                    sizeof(canonicalRegisterDescription));
        AssignString(canonicalMemoryDescription, canonicalMemoryDescription_par,
                    sizeof(canonicalMemoryDescription));
    }
```

Where:

targetName   is the target name.

targetVersion

is the target version.

nrBreakpointsAvailable

is the number of breakpoints available for the interface.

fOverlaySupportAvailable

indicates whether overlays are supported.

fProfilingAvailable

indicates whether profiling is supported for this interface.

nrResetLevels

is the number of reset levels (for example, hard or soft reset). This value must be greater than zero. If it is greater than one, the debugger must obtain a complete list of supported reset levels from the target through CADIExecGetResetLevels().

nrExecModes   is the number of execution modes. If the number of execution modes is greater than two, the debugger must call CADIExecGetModes() to obtain a complete list.

nrExceptions   is the number of exceptions.

nrMemSpaces   is the number of memory spaces.

nrRegisterGroups

is the number of register groups.

nrPipeStages

> is the number of pipeline stages exposed to the debugger. The value can be greater than one only for cycle-accurate models. The value must be one for all other types of model.

nPCRegNum   is the number of the register that is used for the program counter. If there is no program counter available for the target, this value must be set to `CADI_INVALID_REGISTER_ID`.

handledBreakpoints

> indicates the supported breakpoint types. If no breakpoints are supported this is set to 0. Otherwise, this value can be a disjunction of the following values:
> * `CADI_TARGET_FEATURE_BPT_PROGRAM`
> * `CADI_TARGET_FEATURE_BPT_MEMORY`
> * `CADI_TARGET_FEATURE_BPT_REGISTER`
> * `CADI_TARGET_FEATURE_BPT_INST_STEP`
> * `CADI_TARGET_FEATURE_BPT_PROGRAM_RANGE`
> * `CADI_TARGET_FEATURE_BPT_EXCEPTION`.

nrOfHWThreads

> is the number of hardware threads.

nExtendedTargetFeaturesRegNumValid

> indicates whether the extended target features register is supported for registers.

nExtendedTargetFeaturesRegNum

> is the register ID of a string register that contains a static string consisting of colon separated tokens or arbitrary non colon-ASCII char such as `FOO:BAR:ANSWER=42:STARTUP=0xe000`.

> The set and semantics of supported tokens are out of scope of the CADI interface itself. There is no length restriction on this feature string. Having such a string register is optional. Models that do not provide it must set `nExtendedTargetFeaturesRegNumValid` to `false`. In this case, the value of this field must be ignored. Having no such register and having a string register that provides an empty string is equivalent. The following tokens (where *n* denotes a decimal unsigned 32 bit integer) are defined for CADI 2.0:

> PC_MEMSPACE_REGNUM=*n*

>> The ID of the register that contains the current memory space that the program counter points to.

> SP_REGNUM=*n*:

>> The ID of the register used as a stack pointer for the target architecture (or of a register with similar semantics).

> LR_REGNUM=*n*:

>> The ID of the register used as a link register for the target architecture (or of a register with similar semantics).

> STATUS_REGNUM=*n*:

>> The ID of the register used as a status register for the target architecture (or of a register with similar semantics).

> STACK_MEMSPACE_REGNUM=*n*:

>> The ID of the register holding the ID of the memory space currently containing the stack memory.

LOCALVAR_MEMSPACE_REGNUM=*n*:

CADI memory space ID used for local variables. Statically bound to a register that contains the appropriate memspace ID.

GLOBALVAR_MEMSPACE_REGNUM=*n*:

CADI memory space ID used for global vars. Statically bound to a register that contains the appropriate memspace ID.

STACK_MEMSPACE_ID=*n*:

The ID of the memory space that contains the stack.

——— **Note** ———

A model must only expose either STACK_MEMSPACE_ID or STACK_MEMSPACE_REGNUM, that is:

- if the memory space containing the stack is static, then expose STACK_MEMSPACE_ID

- if the memory space containing the stack is expected to change during the course of execution, then expose STACK_MEMSPACE_REGNUM.

LOCALVAR_MEMSPACE_ID=*n*:

The ID of the memory space that is used for storing local variables.

——— **Note** ———

A model must only expose either LOCALVAR_MEMSPACE_ID or LOCALVAR_MEMSPACE_REGNUM, that is:

- if the memory space containing the local variables is static, then expose LOCALVAR_MEMSPACE_ID

- if the memory space containing the local variables is expected to change during the course of execution, then expose LOCALVAR_MEMSPACE_REGNUM.

GLOBALVAR_MEMSPACE_ID=*n*:

The ID of the memory space used for storing global vars.

——— **Note** ———

A model must only expose either GLOBALVAR_MEMSPACE_ID or GLOBALVAR_MEMSPACE_REGNUM, that is:

- if the memory space containing the global variables is static, then expose GLOBALVAR_MEMSPACE_ID

- if the memory space containing the global variables is expected to change during the course of execution, then expose GLOBALVAR_MEMSPACE_REGNUM.

threadID=*s*:

If present, this specifies the name of an implementation-specific mechanism for matching thread-aware breakpoint IDs. One possible value is CONTEXTIDR, which works as described in *Thread-aware breakpoints using CONTEXTIDR* on page B-25.

HALT_CORE=*n*:

The ID of the register used to halt/unclock the current processor. When this register contains 0, the processor executes normally. If a nonzero value is in this register, then the processor is halted and does not execute/step.

If a target does not support one of these features, it does not expose the corresponding token.

canonicalRegisterDescription

is a string that describes the contents of the canonicalRegisterNumber field of CADIRegInfo_t. Canonical register numbers are intended to be target-specific numbers to identify registers in the device by some scheme other than the DWARF index. The format of this field is domain_name/string. The domain_name is that of the organization specifying the scheme. The string part is left to the organization to specify. An example would be arm.com/my/reg/numbers.

canonicalMemoryDescription

is a string that describes the contents of the canonical MemoryNumber field of CADIMemSpaceInfo_t. Canonical memory numbers are intended to be target-specific numbers to identify memory spaces in the device by some scheme other than the DWARF index. The format of this field is 'domain_name/string'. The domain_name is that of the organization specifying the scheme. The string part is specified by the organization such as, for example, arm.com/my/mem/numbers.

canCompleteMultipleInstructionsPerCycle

is true if the target can complete multiple instructions in a single simulation cycle.

B.1.8   CADICallbackType_t

The values in this type identify the different callback functions. These identifiers are, for example, used in the enable vector forwarded to the CADIXfaceAddCallback() call.

Example B-8 CADICallbackType_t

```
enum CADICallbackType_t
{
    CADI_CB_AppliOpen        = 0, // Opens the specified filename and returns a
                                  // streamID that the AppliInput and AppliOutput functions can use.
    CADI_CB_AppliInput       = 1, // This is used for input. Data travels from the host to the target.
    CADI_CB_AppliOutput      = 2, // This is used for output. Data travels from the target to the host.
    CADI_CB_AppliClose       = 3, // Close the stream specified by streamID.
    CADI_CB_String           = 4, // The target system calls this to have the debugger display a
                                  // string. Among other things, it can be used for
                                  // things like hazard and stall indication.
    CADI_CB_ModeChange       = 5, // Call this when the target changes execution modes as defined by
                                  // CADIExecGetModes. The bptNumber parameter is ignored if the mode
                                  // is not CADI_EXECMODE_Bpt.
    CADI_CB_Reset            = 6, // Called when the target is reset.
    CADI_CB_CycleTick        = 7, // This callback, when installed, is
                                  // called after every cycle that is executed by the target.
    CADI_CB_KillInterface    = 8, // This call must ALWAYS be enabled. This is called when the target
                                  // is dying. No further communication with the target is allowed
                                  // after this callback is made.
    CADI_CB_Bypass           = 9, // Callback to bypass the interface, to allow
                                  // any string-based communication with the debugger.
    CADI_CB_LookupSymbol     = 10, //  Lookup a symbol from the debugger.
    CADI_CB_DisasmNotifyModeChange = 11, //  Target mode was changed.
    CADI_CB_DisasmNotifyFileChange = 12, // Target file was changed.
    CADI_CB_Refresh          = 13, // Used to notify debugger that it needs to refresh its state
                                  // (e.g., register values changed)
    CADI_CB_ProfileResourceAccess = 14, //  Profile resource callback.
```

```
  CADI_CB_ProfileRegisterHazard = 15, //  Register hazard callback.
  CADI_CB_Count            = 16,
  CADI_CB_ENUM_MAX = 0xFFFFFFFF    };
```

B.1.9   CADIRefreshReason_t

CADI_REFRESH_REASON_t constants are used by the target to indicate why it has requested a refresh.

Example B-9 CADIRefreshReason_t

```
enum CADIRefreshReason_t
{
  CADI_REFRESH_REASON_MEMORY     = 1,
  CADI_REFRESH_REASON_REGISTERS   = 2,            // also for CADIGetInstructionCount/CADIGetCycleCount
  CADI_REFRESH_REASON_BREAKPOINTS = 4,
  CADI_REFRESH_REASON_PARAMETERS  = 8,
  CADI_REFRESH_REASON_OTHER       = (1 << 31),    // something changed which is not one of the above
  CADI_REFRESH_REASON_ALL         = 0xFFFFFFFF    // all of the above at the same time};
```

B.2 Registers and memory

This section describes data types associated with registers and memory.

B.2.1 CADIReg_t

This data buffer is used to read and write register values. The register data is into the bytes array byte-by-byte. Data is always encoded in little endian mode. For example, the lowest address in the bytes array contains the least significant byte of the register.

Example B-10 CADIReg_t

```
struct CADIReg_t
{
public: // methods
        CADIReg_t(uint32_t regNumber = 0,
                  uint64_t bytes_par = 0,
                  uint16_t offset128 = 0,
                  bool isUndefined = false,
                  CADIRegAccessAttribute_t attribute = CADI_REG_READ_WRITE) :
                regNumber(regNumber), offset128(offset128), isUndefined(isUndefined),
                attribute(attribute)
        {
            for(int i=0; i < 8; ++i)
                bytes[i] = uint8_t(bytes_par >> (i * 8));
        }
public: // data
        uint32_t    regNumber;
        uint8_t     bytes[16];
        uint16_t    offset128;
        bool  isUndefined;
        CADIRegAccessAttribute_t attribute;
};
```

The data members are:

regNumber From debugger to target. Register ID to be read/written.

bytes[16] From target to debugger for reads, from debugger to target for writes. Value to be read/written in little endian (regardless of the endianness of the host or the target).

offset128 From debugger to target. Specify which part of the register value to read/write for long registers greater than 128 bits. Measured in multiples of 128 bits. For example, 1 means bytes[0..15] contain bits 128–255. The actual bitwidth of non-string registers is determined by the bitsWide field in CADIRegInfo_t. Similarly for string registers, specify the offset in units of 16 chars into the string that is to be read or written, for example, offset128=1 means read/write str[16..31]. Reads to offsets beyond the length of the string are explicitly permitted and must result in bytes[0..15] being all zero.

Writes can make the string longer by writing nonzero data to offsets greater than the current length of a string. Writes can make a string shorter by writing data containing at least one zero byte to a specific offset.

Write sequences always write lower offsets before higher offsets and must always be terminated by at least one write containing at least one zero byte. Unused chars in bytes[0..15] (after the terminating zero byte) must be set to zero. The bitsWide field in CADIRegInfo_t is ignored for string registers.

isUndefined   From target to debugger. If true the value of the register is completely undefined. `Bytes[0..15]` must be ignored.

attribute   Undefined for CADI2.0. Targets and Debuggers should not set this data member so that the default value is used.

B.2.2   CADIRegInfo_t

This structure defines information about a register.

Example B-11 CADIRegInfo_t

```
struct CADIRegInfo_t
{
public: // methods
    CADIRegInfo_t(const char *name_par = "",
                const char *description_par = "",
                uint32_t regNumber = 0,
                uint32_t bitsWide = 0,
                int32_t hasSideEffects = 0,
                CADIRegDetails_t details = CADIRegDetails_t(),
                CADIRegDisplay_t display = CADI_REGTYPE_HEX,
                CADIRegSymbols_t symbols = CADIRegSymbols_t(),
                CADIRegFloatFormat_t fpFormat = CADIRegFloatFormat_t(),
                uint32_t lsbOffset = 0, uint32_t dwarfIndex = ~0U,
                bool isProfiled = false, bool isPipeStageField = false,
                uint32_t threadID = 0,
                CADIRegAccessAttribute_t attribute = CADI_REG_READ_WRITE,
                uint32_t canonicalRegisterNumber_ = 0):
        regNumber(regNumber),
        bitsWide(bitsWide),
        hasSideEffects(hasSideEffects),
        details(details),
        display(display),
        symbols(symbols),
        fpFormat(fpFormat),
        lsbOffset(lsbOffset),
        dwarfIndex(dwarfIndex),
        isProfiled(isProfiled),
        isPipeStageField(isPipeStageField),
        threadID(threadID),
        attribute(attribute),
        canonicalRegisterNumber(canonicalRegisterNumber_)
    {
        AssignString(name, name_par, CADI_NAME_SIZE);
        AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
    }

public: // data
        char            name[CADI_NAME_SIZE];
        char            description[CADI_DESCRIPTION_SIZE];
        uint32_t        regNumber;
        uint32_t        bitsWide;
        int32_t         hasSideEffects;
        CADIRegDetails_t  details;
        CADIRegDisplay_t  display;
        CADIRegSymbols_t  symbols;
        CADIRegFloatFormat_t  fpFormat;
        uint32_t        lsbOffset;
        enum { CADI_REGINFO_NO_DWARF_INDEX = 0xffffffff };
        uint32_t        dwarfIndex;
```

```
        bool              isPipeStageField;
        uint32_t          threadID;
        CADIRegAccessAttribute_t attribute;
        uint32_t          canonicalRegisterNumber;

};
```

Where:

name   are the names in the info array.

description are the descriptions in the array.

regNumber   is the register ID. Used by read/write functions to identify the register.

bitsWide   is the bitwidth of non-string register. Ignored for string registers (targets must specify 0 for string registers).

hasSideEffects

             is reserved. Targets must set this to 0 for all registers.

details   is of type `CADIRegDetails_t`, and used to form the Register/SubRegister/SubSubRegister hierarchy. It has two fields:
             • Simple (contains no subregisters).
             • Compound (contains subregisters).

             The two register types work with `CADIRegGetCompound()` *CADI::CADIRegGetCompound()* on page A-28.

display   is the display format. The default is "HEX".

symbols   used for type "symbolic" only.

fpFormat   used for type "float" only.

lsbOffset   is the offset of the least significant bit relative to bit 0 in the parent register (or 0 if there is no parent).

dwarfIndex   is the DWARF register index or `CADI_REGINFO_NO_DWARF_INDEX` if register has no DWARF register index).

isProfiled   indicates that profiling info is available.

isPipeStageField

             is pipe stage field, also true for pc and `contentInfoRegisterId` in `CADIPipeStage_t`.

threadID   is the hardware thread ID, always set to 0.

attribute   are the register access attributes.

canonicalRegisterNumber

             is the canonical register number as defined by the scheme specified in `CADITargetFeatures_t::canonicalRegisterDescription`. If the scheme is the empty string then no meaning can be ascribed to this field.

B.2.3   CADIRegDisplay_t

Register display values. This defines the best way for a debugger to display a register value by default. A debugger can display the value in any format on user request. Only CADI_REGTYPE_STRING is special because in this case the bitsWide field in CADIRegInfo_t is ignored and the debugger retrieves as many ASCII chars until it receives a NUL char.

Example B-12 CADIRegDisplay_t

```
enum CADIRegDisplay_t
{
    CADI_REGTYPE_HEX,       // Hex display (for addresses, etc) - default.
    CADI_REGTYPE_UINT,      // Unsigned integer.
    CADI_REGTYPE_INT,       // Signed integer.
    CADI_REGTYPE_BOOL,      // Boolean (must be one bit).
    CADI_REGTYPE_FLOAT,     // Floating point display (see details).
    CADI_REGTYPE_SYMBOL,    // Symbolic values only.
    CADI_REGTYPE_STRING,    // Strings.
    CADI_REGTYPE_PC,        // Program counter => can be used for disassembly display.
    CADI_REGTYPE_BIN,       // Binary format
    CADI_REGTYPE_OCT        // Octal format
};
```

B.2.4   CADIRegSymbols_t

Array of symbolic values.

Example B-13 CADIRegSymbols_t

```
struct CADIRegSymbols_t
{
public: // methods
        CADIRegSymbols_t(uint32_t numSymbols_par = 0,
                         char **Symbols_par = 0) :
            numSymbols(numSymbols_par),
            Symbols(Symbols_par)
        {
        }
public: // data
        uint32_t    numSymbols;
        char**      Symbols;
};
```

B.2.5   CADIRegAccessAttribute_t

Register access attribute values as listed in Example B-14.

Example B-14 CADIRegAccessAttribute_t

```
enum CADIRegAccessAttribute_t
{
        CADI_REG_READ_WRITE,
        CADI_REG_READ_ONLY,
        CADI_REG_WRITE_ONLY,
```

```
                          CADI_REG_READ_WRITE_RESTRICTED,
                          CADI_REG_READ_ONLY_RESTRICTED,
                          CADI_REG_WRITE_ONLY_RESTRICTED,
                          CADI_REG_ATTR_MAX = 0xffffffff // To force the enum to 32bits, not used
             };
```

B.2.6   CADIRegType_t

Possible register types.

Example B-15 CADIRegType_t

```
enum CADIRegType_t
{
    CADI_REGTYPE_Simple,    // Register which has no subregisters.
    CADI_REGTYPE_Compound   // Register which has subregisters.
};
```

B.2.7   CADIRegDetails_t

Register details structure as listed in Example B-16.

Example B-16 CADIRegDetails_t

```
struct CADIRegDetails_t
{
public: // methods
    CADIRegDetails_t(CADIRegType_t type_par = CADI_REGTYPE_Simple,
                     uint32_t count_par = 0) :
            type(type_par)
        {
            u.compound.count = count_par;
        }
public: // data
        CADIRegType_t type;
        union
        {
            struct
            {
                uint32_t count;
            } compound; //Only valid for CADI_REGTYPE_Compound.
        } u;             // remains a union to leave room for
                         // any other register types we might have
                         // in the future.
};
```

B.2.8   CADIRegGroup_t

Register group description. All fields are target to debugger fields.

```
struct CADIRegGroup_t
{
public: // methods
    CADIRegGroup_t(uint32_t groupID = 0,
                   const char *description_par = "",
                   uint32_t numRegsInGroup = 0,
                   const char *name_par = "",
                   bool isPseudoRegister = false) :
            groupID(groupID), numRegsInGroup(numRegsInGroup),
            isPseudoRegister(isPseudoRegister)
    {
        AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
        AssignString(name, name_par, CADI_NAME_SIZE);
    }

public: // data
    uint32_t  groupID;
    char  description[CADI_DESCRIPTION_SIZE];
    uint32_t  numRegsInGroup;
    char  name[CADI_NAME_SIZE];
    bool  isPseudoRegister;
    };
```

Where:

groupID   is the ID.

description is the total number of registers in the group, including any registers that are not
             direct children of this group.

numRegsInGroup

             is the number of registers in the group.

name   is the group name.

isPseudoRegister

             if true, this register group is not displayed in the register window in the debugger.
             The registers in this group are probably serving other purposes such as pipeline
             stage fields or other special purpose registers such as the PC memory space.

B.2.9   CADIMemSpaceInfo_t

Memory space info data. Each memory space (program and data, for example) in the system has
a separate set of addresses. Any location in the memory of a device can be fully specified with
no less than an indication of the memory space and the address within that space. Only one space
can have the isProgramMemory flag set.

```
struct CADIMemSpaceInfo_t
{
public: // methods
        CADIMemSpaceInfo_t(const char *memSpaceName_par = "",
                           const char *description_par = "",
                           uint32_t memSpaceId = 0,
```

```
                        uint32_t bitsPerMau = 0,
                        CADIAddrSimple_t maxAddress = 0,
                        uint32_t nrMemBlocks = 0,
                        int32_t isProgramMemory = false,
                        CADIAddrSimple_t minAddress = 0,
                        int32_t isVirtualMemory = false,
                        uint32_t isCache = false,
                        uint8_t endianness = 0,
                        uint8_t invariance = 0,
                        uint32_t dwarfMemSpaceId = NO_DWARF_ID) :
            memSpaceId(memSpaceId),
            bitsPerMau(bitsPerMau),
            maxAddress(maxAddress),
            nrMemBlocks(nrMemBlocks),
            isProgramMemory(isProgramMemory),
            minAddress(minAddress),
            isVirtualMemory(isVirtualMemory),
            isCache(isCache),
            endianness(endianness),
            invariance(invariance),
            dwarfMemSpaceId(dwarfMemSpaceId)
    {
        AssignString(memSpaceName, memSpaceName_par, CADI_NAME_SIZE);
        AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
    }
public: // data
        char      memSpaceName[CADI_NAME_SIZE];
        char      description[CADI_DESCRIPTION_SIZE];
    uint32_t   memSpaceId;
    uint32_t   bitsPerMau;
    CADIAddrSimple_t maxAddress;
    uint32_t   nrMemBlocks;
    int32_t    isProgramMemory;
    CADIAddrSimple_t minAddress;
    int32_t    isVirtualMemory;
    uint32_t   isCache;
    uint8_t    endianness;
    uint8_t    invariance;
    enum { NO_DWARF_ID = 0xffffffff };
    uint32_t   dwarfMemSpaceId;
    uint32_t   canonicalMemoryNumber;
};
```

Where:

memSpaceName is the memory space name.

description   is the memory space description.

memSpaceId   is the memory space ID.

bitsPerMau   specifies its per Minimum Addressable Unit (for example, 8 for byte).

maxAddress   is the maximum address of this memory space.

nrMemBlocks   is the number of memory blocks.

isProgramMemory

    specifies program memory. Only one space can have the isProgramMemory flag
    set.

minAddress   specifies the minimum address of this memory space.

isVirtualMemory

        specifies that this memory space is a Virtual/Physical space.

isCache   specifies that this memory space is a cache.

endianness   is the endianness, 0=mono-endian (arch defined), 1=LE, 2=BE .

invariance   is the unit of invariance in bytes, 0=fixed invariance (arch defined).

dwarfMemSpaceId

        is the DWARF memory space ID (`NO_DWARF_ID` if memory space has no DWARF memory space ID).

canonicalMemoryNumber

        is the canonical memory number as defined by the scheme specified in `CADITargetFeatures_t::canonicalMemoryDescription`. If the scheme is the empty string then no meaning can be ascribed to this field.

B.2.10 CADIMemBlockInfo_t

This is a single block of memory addresses (inside a single memory space) that all have the same properties. For example, different memory blocks in the same memory space might be read-only. Blocks can be nested within one another. Blocks at the root level have `CADI_MEMBLOCK_ROOT` as the parent ID.

name is used to give you an idea of the type of memory (`"off chip"`, for example). If cyclesToAccess is 0, the number is unknown or irrelevant.

Example B-19 CADIMemBlockInfo_t

```
struct CADIMemBlockInfo_t
{
public: // methods
    CADIMemBlockInfo_t(const char *name_par = "",
            const char *description_par = "",
            uint16_t id = 0, uint16_t parentID = 0,
            CADIAddrSimple_t startAddr = 0,
            CADIAddrSimple_t endAddr = 0,
            uint32_t cyclesToAccess = 0,
            CADIMemReadWrite_t readWrite = CADI_MEM_ReadWrite,
            uint32_t *supportedMultiplesOfMAU_ = 0,
            uint32_t endianness = 0,
            uint32_t invariance = 0) :
        id(id),
        parentID(parentID),
        startAddr(startAddr),
        endAddr(endAddr),
        cyclesToAccess(cyclesToAccess),
        readWrite(readWrite),
        endianness(endianness),
        invariance(invariance)
    {
        AssignString(name, name_par, CADI_NAME_SIZE);
        AssignString(description, description_par, CADI_DESCRIPTION_SIZE);
        if (supportedMultiplesOfMAU_)
            std::memcpy(supportedMultiplesOfMAU, supportedMultiplesOfMAU_,
                    sizeof(supportedMultiplesOfMAU));
        else
```

```
            std::memset(supportedMultiplesOfMAU, 0, sizeof(supportedMultiplesOfMAU));
        }
public: // data
    char          name[CADI_NAME_SIZE];
    char          description[CADI_DESCRIPTION_SIZE];
    uint16_t      id;
    uint16_t      parentID;
    CADIAddrSimple_t  startAddr;
    CADIAddrSimple_t  endAddr;
    uint32_t      cyclesToAccess;
    CADIMemReadWrite_t readWrite;
    uint32_t      supportedMultiplesOfMAU[CADI_MAU_MULTIPLES_LIST_SIZE];
    uint8_t    endianness;
    uint8_t    invariance;
};
```

Where:

name   is the memory block name.

description   is the memory block description.

id   is the memory block ID.

parentID   The ID of the parent. CADI_MEMBLOCK_ROOT if no parent.

startAddr   is the start address of this memory block.

endAddr   is the end address of this memory block.

cyclesToAccess

                 specifies the number of cycles required for an access to this block.

readWrite   specifies the read/write type of this block.

supportedMultiplesOfMAU

                 indicates the multiples on one byte. If for example the MAU size is 8 bits and the
                 supported access is 32 bits, the corresponding value is 4 (from 32bits/8bits).

endianness   endianness, 0=same as owning memory space, 1=LE, 2=BE.

invariance   is the unit of invariance in bytes, 0=same as owning memory space.

B.2.11   CADIAddr_t

Variables of type CADIAddr_t describe a basic address with the memory space associated with the
address. The definition is as listed in Example B-20:

Example B-20 CADIAddr_t

```
struct CADIAddr_t
{
public: // methods
    CADIAddr_t(CADIMemSpace_t space_par = 0,
              CADIAddrSimple_t addr_par = 0) :
        space(space_par),
        addr(addr_par)
    {
    }
```

```
        bool operator == (const CADIAddr_t &other) const
              { return (space == other.space) && (addr == other.addr); }
public: // data
    CADIMemSpace_t   space;
    CADIAddrSimple_t addr;
};
```

Where:

space   is the numeric designation of the memory space (uint32_t).

addr   is the actual memory address (uint32_t).

B.2.12   CADIMemReadWrite_t

This signifies the read and write status for a block of memory. The definition is as listed in
Example B-21:

Example B-21 CADIMemReadWrite_t

```
enum CADIMemReadWrite_t
{
        CADI_MEM_ReadOnly,
        CADI_MEM_WriteOnly,
        CADI_MEM_ReadWrite,
        CADI_MEM_ENUM_MAX = 0xFFFFFFFF
};
```

B.2.13   CADIAddrComplete_t

Variables of type CADIAddrComplete_t fully specify a single memory location in the target device.
The definition is as listed in Example B-22:

Example B-22 CADIAddrComplete_t

```
struct CADIAddrComplete_t
{
public: // methods
        CADIAddrComplete_t(CADIOverlayId_t overlay_par = 0,
                           CADIAddr_t location_par = CADIAddr_t() ) :
            overlay(overlay_par),
            location(location_par)
        {
        }
        bool operator == (const CADIAddrComplete_t &other)
                          const { return (overlay == other.overlay) &&
                                         (location == other.location); }
public: // data
        CADIOverlayId_t overlay;
        CADIAddr_t      location;
};
```

Where:

overlay   identifies a memory image that can share a region of memory with other memory
             images (uint32_t).

location   is memory address (space ID + numeric address).

B.2.14 CADICacheInfo_t

Cache info data. The definition is as listed in Example B-23:

Example B-23 CADICacheInfo_t

```
struct CADICacheInfo_t
{
public: // methods
    CADICacheInfo_t(uint16_t cacheLineSize_par = 0,
                    uint16_t cacheTagBits_par = 0,
                    uint16_t associativity_par = 0,
                    bool writeThrough_par = false) :
        cacheLineSize(cacheLineSize_par),
        cacheTagBits(cacheTagBits_par),
        associativity(associativity_par),
        writeThrough(writeThrough_par)
        {
        }
public: // data
    uint16_t      cacheLineSize;
    uint16_t      cacheTagBits;
    uint16_t      associativity;
    bool          writeThrough;
};
```

Where:

cacheLineSize

   is the size of a cacheline in bytes.

cacheTagBits is the size of a tag in bits.

associativity

   is 1,2,4, or 8-way associative.

writeThrough if true, the dirty flag is not used.

B.3 Breakpoints and execution control

This section describes data types associated with breakpoints and control of the application running on the target.

B.3.1 CADIBptRequest_t

The breakpoint request provides the PC address at which a breakpoint must occur and a string that describes the condition of the breakpoint. The target decides whether it can implement the breakpoint conditions.

Example B-24 CADIBptRequest_t

```
struct CADIBptRequest_t
{
public: // methods
        CADIBptRequest_t(const CADIAddrComplete_t address = CADIAddrComplete_t(),
                        uint64_t sizeOfAddressRange=0,
                        int32_t enabled=0,
                        const char *conditions_par = "",
                        bool useFormalCondition = 1,
                        CADIBptCondition_t formalCondition = CADIBptCondition_t(),
                        CADIBptType_t type = CADI_BPT_PROGRAM,
                            uint32_t regNumber = 0,
                        int32_t temporary = false,
                        uint8_t triggerType = 0,
                        uint32_t continueExecution = false) :
            address(address),
            sizeOfAddressRange(sizeOfAddressRange),
            enabled(enabled),
            useFormalCondition(useFormalCondition),
            formalCondition(formalCondition), type(type),
            regNumber(regNumber),
            temporary(temporary),
            triggerType(triggerType),
            continueExecution(continueExecution)
        {
            AssignString(conditions, conditions_par, CADI_DESCRIPTION_SIZE);
        }
public: // data
        CADIAddrComplete_t  address;
        uint64_t            sizeOfAddressRange;
        int32_t             enabled;
        char                conditions[CADI_DESCRIPTION_SIZE];
        bool                useFormalCondition;
        CADIBptCondition_t  formalCondition;
        CADIBptType_t       type;
        uint32_t            regNumber;
        int32_t             temporary;
        uint8_t             triggerType;
        uint32_t            continueExecution;
    };
```

Where:

address   is the PC address at which the breakpoint is to occur.

sizeOfAddressRange

            is used only if type is CADI_BPT_PROGRAM_RANGE.

enabled   selects Enable/Disable breakpoint.

conditions   are the breakpoint conditions. Ultimately the target decides if it can implement breakpoint conditions.

useFormalCondition

if 0, use free-form `conditions`. If 1, use `formalCondition`.

formalCondition

are the formal conditions.

type   is the type.

regNumber   is only used for the register type.

temporary   specifies a temporary breakpoint.

triggerType   enables breakpoints that trigger only on read, write, or modify of the register or memory. Use the following defines to set the trigger:

- `CADI_BPT_TRIGGER_ON_READ` triggers a breakpoint if the associated memory or register is read from by either a normal or debug read.

- `CADI_BPT_TRIGGER_ON_WRITE` triggers a breakpoint if the associated memory or register is written to by either a normal or debug read.

- `CADI_BPT_TRIGGER_ON_MODIFY` triggers a breakpoint if the value of the associated register/memory has been modified. This might be the result of an explicit register/memory access or (for example in case of registers/memory-mapped registers) of executing an instruction.

The trigger condition defines can be ORed together to make the breakpoint sensitive to more than one condition.

————— **Note** —————

`triggerType` only has meaning for `CADI_BPT_REGISTER` and `CADI_BPT_MEMORY` breakpoints:

- The debugger must set this to zero for other breakpoint types.

- Setting `triggerType` to zero for `CADI_BPT_REGISTER` and `CADI_BPT_MEMORY` results in undefined behavior and must not be done.

———————————————

continueExecution

if 1, continue execution after breakpoint has been hit. This field must be obeyed by all types of breakpoints, including `CADI_BPT_INST_STEP`.

B.3.2   CADIBptCondition_t and CADIBptConditionOperator_t

Breakpoint comparison operations only apply to `CADI_BPT_MEMORY` and `CADI_BPT_REGISTER` breakpoints. Other breakpoints must always specify `CADI_BPT_COND_UNCONDITIONAL` as `conditionOperator`. Breakpoint conditions are always applied as a secondary condition after the primary condition of the breakpoint that depends on the breakpoint type and the trigger type.

`CADI_BPT_PROGRAM`, `CADI_BPT_PROGRAM_RANGE`, `CADI_BPT_INST_STEP`, `CADI_BPT_EXCEPTION` must obey the `ignoreCount` if the `useFormalCondition` is set. However, the debugger must ensure that `conditionOperator` is `CADI_BPT_COND_UNCONDITIONAL`, otherwise the behavior is undefined.

```
struct CADIBptCondition_t
{
public: // methods
    CADIBptCondition_t(
                    CADIBptConditionOperator_t conditionOperator = CADI_BPT_COND_UNCONDITIONAL,
                    int64_t comparisonValue = 0,
                    uint32_t threadID = 0,
                    uint32_t ignoreCount = 0,
                    uint32_t bitwidth = 0) :
            conditionOperator(conditionOperator),
            comparisonValue(comparisonValue),
            threadID(threadID),
            ignoreCount(ignoreCount),
            bitwidth(bitwidth)
    {
    }
public: // data
    CADIBptConditionOperator_t    conditionOperator;
    int64_t                       comparisonValue;
    uint32_t                      threadID;
    uint32_t                       ignoreCount;
    uint32_t                       bitwidth;
};
```

Where:

conditionOperator

> specifies the types of condition that are used to determine whether a breakpoint matches. This specifies how the fields comparisonValue and threadID are interpreted. Targets must return CADI_STATUS_ArgNotSupported from CADIBptSet() if this field is set to a value that is not supported by the target.

comparisonValue

> if the bottom 30-bits of conditionOperator have a value other than CADI_BPT_COND_UNCONDITIONAL (0), then they specify how comparisonValue is to be compared with the value associated with the breakpoint hit. See the list of condition values enumerated in Example B-26 on page B-25.

threadID   if bit 31 of conditionOperator is set (CADI_BPT_COND_THREADID), then this field specifies that the breakpoint only hits if threadID matches the current thread. It is up to the target to specify how thread IDs are to be matched. Targets must use the extendedTargetFeatures register to identify the mechanism being used. (See *CADITargetFeatures_t structure* on page B-5 under the nExtendedTargetFeaturesRegNum entry.) One possible mechanism for ARM targets is to match against CONTEXTIDR, as described in *Thread-aware breakpoints using CONTEXTIDR* on page B-25. Targets must return CADI_STATUS_ArgNotSupported from CADIBptSet() if threadID is nonzero and bit 31 of conditionOperator is not set.

ignoreCount   is the number of breaks to ignore.

bitwidth   is the width of comparison value.

The conditional breakpoint operations are enumerated in CADIBptConditionOperator_t as listed in Example B-26 on page B-25.

```
enum CADIBptConditionOperator_t
{
    CADI_BPT_COND_UNCONDITIONAL,    // Normal breakpoint, always break, no additional condition.
    CADI_BPT_COND_EQUALS,           // Only break if value == comparisionValue (unsigned comparison)
    CADI_BPT_COND_NOT_EQUALS,       // Only break if value != comparisionValue (unsigned comparison)
        // signed comparison
    CADI_BPT_COND_GREATER_THAN_SIGNED,           // Only break if value >  comparisionValue
    CADI_BPT_COND_GREATER_THAN_OR_EQUALS_SIGNED, // Only break if value >= comparisionValue
    CADI_BPT_COND_LESS_THAN_SIGNED,              // Only break if value <  comparisionValue
    CADI_BPT_COND_LESS_THAN_OR_EQUALS_SIGNED,    // Only break if value <= comparisionValue
        // unsigned comparison
    CADI_BPT_COND_GREATER_THAN_UNSIGNED,           // Only break if value >  comparisionValue
    CADI_BPT_COND_GREATER_THAN_OR_EQUALS_UNSIGNED, // Only break if value >= comparisionValue
    CADI_BPT_COND_LESS_THAN_UNSIGNED,              // Only break if value <  comparisionValue
    CADI_BPT_COND_LESS_THAN_OR_EQUALS_UNSIGNED,    // Only break if value <= comparisionValue
    CADI_BPT_COND_ENUM_COUNT,                      // Not a valid condition operator
        // legacy support, same as signed comparison
    CADI_BPT_COND_GREATER_THAN = CADI_BPT_COND_GREATER_THAN_SIGNED,
    CADI_BPT_COND_GREATER_THAN_OR_EQUALS = CADI_BPT_COND_GREATER_THAN_OR_EQUALS_SIGNED,
    CADI_BPT_COND_LESS_THAN = CADI_BPT_COND_LESS_THAN_SIGNED,
    CADI_BPT_COND_LESS_THAN_OR_EQUALS = CADI_BPT_COND_LESS_THAN_OR_EQUALS_SIGNED,
        // these are no breakpoint conditions:
    CADI_BPT_COND_ENUM_MAX = 0xFFFFFFFF
    CADI_BPT_COND_LESS_THAN_OR_EQUALS = CADI_BPT_COND_LESS_THAN_OR_EQUALS_SIGNED,
    CADI_BPT_COND_THREADID = 0x80000000,    // Thread-aware breakpoint
        // these are no breakpoint conditions:
    CADI_BPT_COND_ENUM_MAX = 0xFFFFFFFF
};
```

Thread-aware breakpoints using CONTEXTIDR

ARM targets support thread-aware breakpoints by matching the threadID field against the 32-bit CONTEXTIDR register in the target. Targets must indicate support for this mechanism by including the string threadID=CONTEXTIDR in the extendedTargetFeatures register. See *CADITargetFeatures_t structure* on page B-5 under the nExtendedTargetFeaturesRegNum entry.

Using this mechanism, whenever a breakpoint condition is met and bit 31 of conditionOperator field is set, the threadID field is compared against CONTEXTIDR. If threadID and CONTEXTIDR are equal, the breakpoint hits. If they differ, the breakpoint does not hit and is ignored. The threadID field is ignored if bit 31 of conditionOperator is 0.

B.3.3 CADIBptDescription_t

CADIBptDescription_t is listed in Example B-27:

```
struct CADIBptDescription_t
{
public: // methods
    CADIBptDescription_t(CADIBptNumber_t bptNumber_par = 0,
                         CADIBptRequest_t bptInfo_par = CADIBptRequest_t()) :
            bptNumber(bptNumber_par),
            bptInfo(bptInfo_par)
        {
```

```
        }
public: // data
    CADIBptNumber_t    bptNumber;
    CADIBptRequest_t  bptInfo;
};
```

Where:

bptNumber   is the breakpoint number (`uint32_t`).

bptInfo   is the breakpoint information such as address or condition.

B.3.4   CADIBptConfigure_t

The definition of CADIBptConfigure_t is listed in Example B-28:

Example B-28 CADIBptConfigure_t

```
enum CADIBptConfigure_t
  {
    CADI_BPT_Disable,
    CADI_BPT_Enable
  };
```

B.3.5   CADIExecMode_t structure

This structure is used to return the execution mode.

Example B-29 CADIExecMode_t structure

```
struct CADIExecMode_t
{
public:
    CADIExecMode_t(uint32_t number = 0,
          const char *name_par = "") :
          number(number)
    {
       AssignString(name, name_par, CADI_NAME_SIZE);
    }
    uint32_t   number;
    char   name[CADI_NAME_SIZE];
}
```

Where:

number   indicates the execution mode and must be one of the values listed in
               CADI_EXECMODE_t enumeration.

name   is the mode name.

B.3.6   CADI_EXECMODE_t enumeration

The values in CADI_EXECMODE_t enumeration are listed in Example B-30 on page B-27:

```
enum CADI_EXECMODE_t {
    CADI_EXECMODE_Stop = 0,
    CADI_EXECMODE_Run = 1,
    CADI_EXECMODE_Bpt = 2,
    CADI_EXECMODE_Error = 3,
    CADI_EXECMODE_HighLevelStep = 4,       // Reserved for future use.
    CADI_EXECMODE_RunUnconditionally = 5, // Reserved for future use.
    CADI_EXECMODE_ResetDone = 5,
    CADI_EXECMODE_ENUM_MAX = 0xFFFFFFFF    };
```

The enumeration values are used by modeChange():

modeChange(CADI_EXECMODE_Stop)

> The simulation was in state running and has now stopped. This is always the last callback in a sequence of callbacks when the simulation stopped. If the stop was because one or more breakpoints have been hit then this callback is preceded by one or more modeChange(CADI_EXECMODE_Bpt, num) callbacks where num specified the breakpoint(s) being hit. CADIExecStop() eventually results in a modeChange(CADI_EXECMODE_Stop) callback. This callback implies a refresh(REGISTERS|MEMORY) callback that indicates a debugger must assume registers and memory have changed.

modeChange(CADI_EXECMODE_Run)

> The simulation was in state stopped and is now running. CADIExecContinue() and CADIExecSingleStep() eventually result in a modeChange(CADI_EXECMODE_Run) callback.

modeChange(CADI_EXECMODE_Bpt, num)

> The breakpoint number num of the breakpoint being hit is passed as the second parameter in the modeChange callback. This callback can be called several times in a straight sequence if multiple breakpoints have been hit at the same time. A modeChange(CADI_EXECMODE_Stop) callback is always following and terminating this sequence, except if continueExecution was true for all breakpoints being hit.

> ──── **Note** ────
>
> This callback does not mean that the simulation stopped. It can be followed by more modeChange(CADI_EXECMODE_Bpt, num) callbacks. The final modeChange(CADI_EXECMODE_Stop) is responsible for signaling that the simulation stopped.
> ────────

modeChange(CADI_EXECMODE_Error)

> This is the same as modeChange(CADI_EXECMODE_Stop), but the model is in a state stopped and error after this callback. This means that all execution control functions are disabled. CADIExecReset() must be called first to enable them again. This callback is not followed by another modeChange(CADI_EXECMODE_Stop) callback, it implies modeChange(CADI_EXECMODE_Stop). This callback implies a refresh(REGISTERS|MEMORY) callback which means that a debugger must assume registers and memory have changed.

modeChange(CADI_EXECMODE_ResetDone)

> The CADIExecReset() request recently requested by a debugger is now complete. This is always the last callback in a sequence of callbacks cased by a CADIExecReset(). A modeChange(CADI_EXECMODE_Stop) might happen before this callback if the model was running when CADIExecReset() was issued.

> CADIExecReset() is an asynchronous call. Each debugger connected to a target, including the caller, receives this callback after the simulation finishes the reset.

> This callback implies a refresh(REGISTERS|MEMORY) callback that indicates that a debugger must assume registers and memory have changed.

B.3.7 CADIResetLevel_t

The definition of CADIResetLevel_t is:

Example B-31 CADIResetLevel_t

```
struct CADIResetLevel_t
{
public: // methods
    CADIResetLevel_t(uint32_t number_par = 0,
                     const char *name_par = "") :
        number(number_par)
    {
        AssignString(name, name_par, sizeof(name));
    }
public: // data
    uint32_t   number;
    char   name[CADI_NAME_SIZE];
};
```

B.3.8 CADIException_t

The definition of CADIException_t is as listed in Example B-32:

Example B-32 CADIException_t

```
struct CADIException_t
{
public: // methods
    CADIException_t(uint32_t number_par = 0,
                const char *name_par = "",
                CADIAddr_t vector_par = CADIAddr_t()) :
        number(number_par),
        vector(vector_par)
    {
        AssignString(name, name_par, sizeof(name));
    }
public: // data
    uint32_t   number;
    char   name[CADI_NAME_SIZE];
    CADIAddr_t vector;
};
```

B.3.9   CADIExceptionAction_t

The definition of `CADIExceptionAction_t` is as listed in Example B-33:

Example B-33 CADIExceptionAction_t

```
// Exception action data
enum CADIExceptionAction_t
{
    CADI_EXCEPTION_Raise,        ///< For targets that can raise an exception
    CADI_EXCEPTION_Lower,        ///< ... and leave it raised
    CADI_EXCEPTION_Pulse,
    CADI_EXCEPTION_ENUM_MAX = 0xFFFFFFFF
};
```

B.4 Pipelines

This section describes data types associated with instruction pipelines.

B.4.1 CADIPipeStage_t

An object of type `CADIPipeStage_t` describes a single pipe stage. The definition is as listed in Example B-34:

Example B-34 CADIPipeStage_t

```
struct CADIPipeStage_t
{
public: // methods
    CADIPipeStage_t(uint32_t id_par = 0, const char *name_par = "",
                    uint32_t pc_par = CADI_INVALID_REGISTER_ID,
                    uint32_t contentInfoRegisterId_par = CADI_INVALID_REGISTER_ID) :
            id(id_par), pc(pc_par),
            contentInfoRegisterId(contentInfoRegisterId_par)
        {
            AssignString(name, name_par, sizeof(name));
        }
public: // data
    uint32_t   id;
    char   name[CADI_NAME_SIZE];
    uint32_t   pc;
    uint32_t contentInfoRegisterId;
};
```

Where:

`id`	is the ID.
`name`	is the stage name.
`pc`	is the register ID that holds the address of the instruction.
`contentInfoRegisterId`	

is the register id that holds the current content info for this pipe stage. The values of this register correspond to the `CADIPipeStageContentInfo_t` enumeration.

B.4.2 CADIPipeStageContentInfo_t

The definition of `CADIPipeStageContentInfo_t` is as listed in Example B-35:

Example B-35 CADIPipeStageContentInfo_t

```
enum CADIPipeStageContentInfo_t
{
  CADI_PIPESTAGE_Invalid,         // This pipe stage is empty or invalid,  nothing is displayed.
  CADI_PIPESTAGE_OpcodeOnly,      // An instruction is in this stage, only the opcode is valid.
  CADI_PIPESTAGE_DisassemblyOnly, // An instruction is in this stage, only the disassembly is valid.
  CADI_PIPESTAGE_Instruction,     // An instruction is in this stage, both the
                                  // opcode and the disassembly are valid.
  CADI_PIPESTAGE_ENUM_COUNT,
  CADI_PIPESTAGE_MAX = 0xFFFFFFFF
};
```

B.5 Disassembly

This section describes data types associated with disassembly of the application code running on the target.

B.5.1 CADIDisassemblerStatus

The `CADIDisassemblerStatus` enumeration is as listed in Example B-36:

Example B-36 CADIDisassemblerStatus

```
enum CADIDisassemblerStatus
{
   CADI_DISASSEMBLER_STATUS_OK,              // disassembling completed successfully
   CADI_DISASSEMBLER_STATUS_NO_INSTRUCTION,  // current address points to illegal instructions/data
   CADI_DISASSEMBLER_STATUS_ILLEGAL_ADDRESS, // address out of range (memory read failed)
   CADI_DISASSEMBLER_STATUS_ERROR            // other error
};
```

B.5.2 CADIDisassemblerType

The `CADIDisassemblerType` enumeration s as listed in Example B-37:

Example B-37 CADIDisassemblerType

```
enum CADIDisassemblerType
{
   CADI_DISASSEMBLER_TYPE_STANDARD,      // disassembly supporting a PC and lookahead
   CADI_DISASSEMBLER_TYPE_SOURCELEVEL=2, // source level assembly / C
   CADI_DISASSEMBLER_TYPE_INTERPRETER    // interpreter window (e.g. for scripts)};
```

B.5.3 CADIDisassemblerInstructionType

The `CADIDisassemblerInstructionType` enumeration is as listed in Example B-38:

Example B-38 CADIDisassemblerInstructionType

```
enum CADIDisassemblerInstructionType
{
   CADI_DISASSEMBLER_INSTRUCTION_TYPE_NOCALL,  // The instruction is not a call, so for example an ALU
                                               // instruction, memory access, or a jump
   CADI_DISASSEMBLER_INSTRUCTION_TYPE_CALL     // The instruction is a call into a subroutine.
                          //Program  flow is expected to return after the subroutine has finished.
};
```

B.6 Semihosting and message output

This section describes data types related to semihosting and message output.

B.6.1 CADISemiHostingInputChannelType_t

Reverse semihosting (for interrupts from the debugger towards the target).

Example B-39 CADISemiHostingInputChannelType_t

```
enum CADISemiHostingInputChannelType_t
{
    CADI_INPUT_KEYBOARD,
    CADI_INPUT_POINTING_DEVICE
};
```

B.6.2 CADISemiHostingInputChannel_t

Reverse semihosting (for interrupts from the debugger towards the target).

Example B-40 CADISemiHostingInputChannel_t

```
struct CADISemiHostingInputChannel_t{public: // methods
    CADISemiHostingInputChannel_t(uint32_t ID_par = 0,
                        const char *name_par = "",
                        CADISemiHostingInputChannelType_t type_par = CADI_INPUT_KEYBOARD) :
            ID(ID_par), type(type_par)
        {
            AssignString(name, name_par, sizeof(name));
        }
public: // data
    uint32_t ID;
    char name[CADI_NAME_SIZE];
    CADISemiHostingInputChannelType_t type;
};
```

B.6.3 CADIConsoleChannel_t

Reverse semihosting for interrupts from the debugger towards the target.

Example B-41 CADIConsoleChannel_t

```
struct CADIConsoleChannel_t{public: // methods
    CADIConsoleChannel_t(uint32_t streamID_par,
                        const char *name_par = "",
                        bool blocking_par = false,
                        bool characterInput_par = false) :
            streamID(streamID_par),
            blocking(blocking_par),
            characterInput(characterInput_par)
        {
            AssignString(name, name_par, sizeof(name));
        }
public: // data
```

```
    uint32_t streamID;
    char name[CADI_NAME_SIZE];
    bool blocking;
    bool characterInput;
};
```

Where:

streamID   is the stream identifier.

name   is the stream name.

blocking   if true, the console is blocking for the appliInput() function.

characterInput

> if true, then the notify/return from call is on a per character basis. If false, then the notify/return from call is on a or per line basis.

B.6.4 CADIStreamId

This set of streamIds is reserved for the following special cases:

```
CADICallbackObj::appliInput( uint32_t, uint32_t, uint32_t*, char*)
CADICallbackObj::appliOutput( uint32_t, uint32_t, uint32_t*, char const*)
```

They automatically exist and no special action is required to use them. Attempting to CADICallbackObj::appliClose(uint32_t) these handles results in undefined behavior and must not be done.

Example B-42 CADIStreamId

```
enum CADIStreamId{
        CADI_STREAMID_STDIN = 0,
        CADI_STREAMID_STDOUT = 1,
        CADI_STREAMID_STDERR = 2
};
```

B.7 Profiling and tracing

This section describes data types associated with profiling and tracing.

B.7.1 CADIProfileResultType_t

The `CADIProfileResultType_t` enumeration enables the target to specify whether the results represent a percentage of the whole or a total count:

Example B-43 CADIProfileResultType_t

```
enum CADIProfileResultType_t
    {
        CADI_PROF_RESULT_Percentage,
        CADI_PROF_RESULT_Count
    };
```

B.7.2 CADIProfileResults_t

An object of this type contains the results of a profiling session.

Example B-44 CADIProfileResults_t

```
class CADIProfileResults_t
    public: // methods
        CADIProfileResults_t(uint32_t regionNumber_par = 0,
                             uint32_t accesses_par = 0) :
            regionNumber(regionNumber_par),
            accesses(accesses_par)
        {
        }
    public: // data
        uint32_t   regionNumber;
        uint32_t   accesses;
    };
```

B.7.3 CADIProfileRegion_t

Objects of this type describe a memory range to be profiled. A region is part of a group of one or more regions. If `addressesAreValid` is not `true`, then the object refers to the entire memory space that is not included by another region.

--- **Note** ---

Two overlays for the same memory addresses do not constitute a shared memory space.

The definition of `CADIProfileRegion_t` s as listed in Example B-45:

Example B-45 CADIProfileRegion_t

```
class CADIProfileRegion_t
    {
    public: // methods
```

```
                        CADIProfileRegion_t(int32_t addressesAreValid_par = false,
                                    CADIOverlayId_t overlay_par = 0,
                                    CADIMemSpace_t memorySpace_par = 0,
                                    CADIAddrSimple_t start_par = 0,
                                    CADIAddrSimple_t finish_par = 0) :
                            addressesAreValid(addressesAreValid_par),
                            overlay(overlay_par),
                            memorySpace(memorySpace_par),
                            start(start_par),
                            finish(finish_par)
            {
            }
    public: // data
    int32            addressesAreValid;
    CADIOverlayId_t   overlay;
    CADIMemSpace_t    memorySpace;
    CADIAddrSimple_t  start;
    CADIAddrSimple_t  finish;
};
```

B.7.4   CADIProfileType_t

The `CADIProfileType_t` enumeration determines the type of profiling to which the region definition applies.

Example B-46 CADIProfileType_t

```
enum CADIProfileType_t
    {
        CADI_PROF_TYPE_Execution,
        CADI_PROF_TYPE_Memory,  // Used with CADIProfileGetMemory.
        CADI_PROF_TYPE_Trace    // Used with CADIProfileGetTrace.
    };
```

B.7.5   CADIProfileControl_t

The `CADIProfileControl_t` enumeration is used to describe the action the call is trying to apply to the target profiling mechanism.

Example B-47 CADIProfileControl_t

```
enum CADIProfileControl_t
    {
        CADI_PROF_CNTL_Start,
        CADI_PROF_CNTL_Stop,
        CADI_PROF_CNTL_Reset
    };
```

B.7.6   CADIRegProfileResults_t

Objects of this type hold access information for a register.

```
class CADIRegProfileResults_t
{
public: // methods
        CADIRegProfileResults_t(uint32_t regID_par = 0,
                                uint64_t readAccesses_par = 0,
                                uint64_t writeAccesses_par = 0) :
                regID(regID_par), readAccesses(readAccesses_par),
writeAccesses(writeAccesses_par)
            {
            }
public: // data
        uint32_t   regID;
        uint64_t   readAccesses;
        uint64_t   writeAccesses;
};
```

B.7.7   CADIMemProfileResults_t

Objects of this type hold access information for a memory range.

```
class CADIMemProfileResults_t
{
public: // methods
        CADIMemProfileResults_t(CADIAddrSimple_t address_par = 0,
                                uint64_t readAccesses_par = 0,
                                uint64_t writeAccesses_par = 0) :
            address(address_par),
            readAccesses(readAccesses_par),
            writeAccesses(writeAccesses_par)
        {
        }
public: // data
        CADIAddrSimple_t address;
        uint64_t   readAccesses;
        uint64_t   writeAccesses;
};
```

B.7.8   CADIInstructionProfileResults_t

Objects of this type hold execution information for an instruction.

```
class CADIInstructionProfileResults_t
{
public: // methods
      CADIInstructionProfileResults_t(uint32_t FID_par = 0,
                                const char *name_par = "",
                                const char *pathToInstructionInLISASource_par = "",
                                uint64_t executionCount_par = 0) :
          FID(FID_par),
```

```
        executionCount(executionCount_par)
     {
         AssignString(name, name_par, sizeof(name));
         AssignString(pathToInstructionInLISASource, pathToInstructionInLISASource_par,
                      sizeof(pathToInstructionInLISASource_par));
     }
public: // data
     uint32_t FID;
     char    name[CADI_DESCRIPTION_SIZE];
     char    pathToInstructionInLISASource[CADI_DESCRIPTION_SIZE];
     uint64_t executionCount;
};
```

B.7.9   CADIProfileResourceAccessType_t

This defines the accesses that are permitted for the resource.

Example B-51 CADIProfileResourceAccessType_t

```
enum CADIProfileResourceAccessType_t
{
        CADI_PROF_ACCESS_READ,
        CADI_PROF_ACCESS_WRITE,
        CADI_PROF_ACCESS_READ_OR_WRITE
};
```

B.7.10   CADIProfileHazardTypes_t

This defines hazard information for the resource.

Example B-52 CADIProfileHazardTypes_t

```
enum CADIProfileHazardTypes_t
{
        CADI_PROF_HAZARD_RESOURCE_MAX_ACCESS,
        CADI_PROF_HAZARD_RESOURCE_MIN_ACCESS,
        CADI_PROF_HAZARD_RESOURCE_MAX_WRITE_ACCESS,
        CADI_PROF_HAZARD_RESOURCE_MAX_READ_ACCESS,
        CADI_PROF_HAZARD_RESOURCE_READ_AFTER_WRITE,
        CADI_PROF_HAZARD_RESOURCE_WRITE_AFTER_READ,
        CADI_PROF_HAZARD_CONTROL,
        CADI_PROF_HAZARD_OTHER
};
```

B.7.11   CADIProfileHazardDescription_t

Objects of this type provide information about the hazard.

Example B-53 CADIProfileHazardDescription_t

```
class CADIProfileHazardDescription_t
{
public: // methods
```

```
                            CADIProfileHazardDescription_t(
                                    CADIProfileHazardTypes_t type_par =
                                            CADI_PROF_HAZARD_RESOURCE_MAX_ACCESS,
                                    uint32_t numberOfAccesses_par = 0,
                                    uint32_t originInstructionFID_par = 0,
                                    uint32_t affectedInstructionFID_par = 0,
                                    const char *resource_par = "",
                                    const char *messages_par = "") :
                            type(type_par),
                            numberOfAccesses(numberOfAccesses_par),
                            originInstructionFID(originInstructionFID_par),
                            affectedInstructionFID(affectedInstructionFID_par)
                    {
                        AssignString(resource, resource_par, sizeof(resource));
                        AssignString(message, messages_par, sizeof(message));
                    }
public: // data
                    CADIProfileHazardTypes_t type;
                    uint32_t    numberOfAccesses;
                    uint32_t    originInstructionFID;
                    uint32_t    affectedInstructionFID;
                    char        resource[CADI_DESCRIPTION_SIZE];
                    char        message[CADI_DESCRIPTION_SIZE];
};
```

Where:

type   is the number of accesses to affected resource.

numberOfAccesses

                is the FID of the originator resource/instruction.

affectedInstructionFID

                is the name of the affected resource/instruction.

resource   is the resource.

message   is the hazard message.

B.7.12   CADITraceControl_t

This describes the type of control being exerted on the trace mechanism. `CADITraceControl_t` is defined as listed in Example B-54:

Example B-54 CADITraceControl_t

```
enum CADITraceControl_t
    {
        CADI_TRACE_CNTL_StartContinuous,
        CADI_TRACE_CNTL_StartDiscontinuity,
        CADI_TRACE_CNTL_Stop
    };
```

B.7.13   CADITraceBufferControl_t

This describes the type of control being exerted on the trace mechanism.

Example B-55 CADITraceBufferControl_t

```
enum CADITraceBufferControl_t
    {
        CADI_TRACE_BUFF_Wrap,
        CADI_TRACE_BUFF_StopOnFull
    };
```

B.7.14 CADITraceOverlayControl_t

This describes the type of control being exerted on the trace mechanism.

Example B-56 CADITraceOverlayControl_t

```
enum CADITraceOverlayControl_t
    {
        CADI_TRACE_OVERLAY_Manager,
        CADI_TRACE_OVERLAY_Memory
    };
```

B.7.15 CADITraceBlockType_t

This describes the type of data in a CADITraceBlock_t.

Example B-57 CADITraceBlock_t

```
enum CADITraceBlockType_t
    {
        CADI_TRACE_BLK_Address,
        CADI_TRACE_BLK_Overlay
    };
```

B.7.16 CADITraceBlock_t

This describes a single piece of trace data that either contains an overlay ID or an address.

Example B-58 CADITraceBlock_t

```
struct CADITraceBlock_t
{
public: // methods
        CADITraceBlock_t(CADITraceBlockType_t blockType_par = CADI_TRACE_BLK_Address,
                        CADIAddr_t address_par = CADIAddr_t(),
                        CADIOverlayId_t overlay_par = CADIOverlayId_t()) :
            blockType(blockType_par)
        {
            u.address = address_par;
            u.overlay = overlay_par;
        }
public: // data
        CADITraceBlockType_t blockType;
        struct
```

```
        {
            CADIAddr_t      address;
            CADIOverlayId_t overlay;
        } u;
};
```