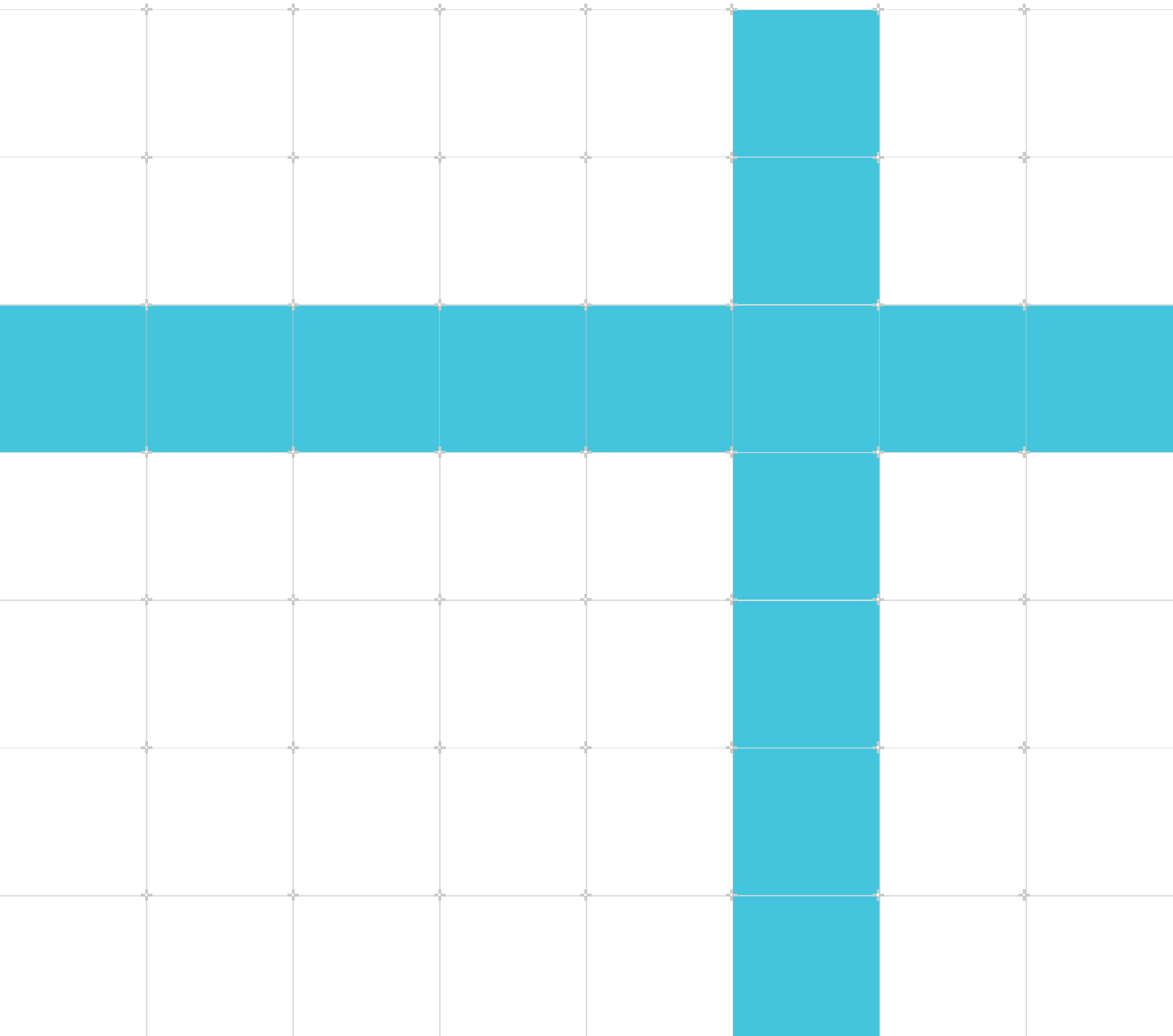
Component Architecture Debug Interface

Version 2.0

User Guide

Issue 13
100963_0200_13_en



Component Architecture Debug Interface
User Guide

Release Information

Document history

Issue	Date	Confidentiality	Change
0200-00	31 May 2017	Non-Confidential	Update for v11.0. Document numbering scheme has changed.
0200-01	31 August 2017	Non-Confidential	Update for v11.1.
0200-02	17 November 2017	Non-Confidential	Update for v11.2.
0200-03	23 November 2018	Non-Confidential	Update for v11.5.
0200-04	26 February 2019	Non-Confidential	Update for v11.6.
0200-05	17 May 2019	Non-Confidential	Update for v11.7.
0200-06	12 March 2020	Non-Confidential	Update for v11.10.
0200-07	22 September 2020	Non-Confidential	Update for v11.12.
0200-08	9 December 2020	Non-Confidential	Update for v11.13.
0200-09	29 June 2021	Non-Confidential	Update for v11.15.
0200-10	15 June 2022	Non-Confidential	Update for v11.18.
0200-11	14 September 2022	Non-Confidential	Update for v11.19.
0200-12	7 December 2022	Non-Confidential	Update for v11.20.

Issue	Date	Confidentiality	Change
0200-13	13 September 2023	Non-Confidential	Update for v11.23.

Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. All rights reserved. Other brands and

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

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.

1.1 Conventions

The following subsections describe conventions used in Arm documents.

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 the Arm Glossary for more information: developer.arm.com/glossary.

Convention	Use
italic	Citations.
bold	Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses 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	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

Recommendations. Not following these recommendations might lead to system failure or damage.

Requirements for the system. Not following these requirements might result in system failure or damage.

Requirements for the system. Not following these requirements will result in system failure or damage.

An important piece of information that needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

A reminder of something important that relates to the information you are reading.

1.2 Other information

See the Arm® website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

1.3 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm® product resources	Document ID	Confidentiality
Fast Models Reference Guide	100964	Non-Confidential
Iris User Guide	101196	Non-Confidential

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com.

2. Introduction to the Component Architecture Debug Interface (CADI)

CADI is a C++ API that enables convenient and accurate debugging of complex System on Chip (SoC) simulation platforms.

CADI is deprecated and is being replaced by the Iris debug interface. Support for CADI will be removed in a future Fast Models release. See the Fast Models release notes for the expected removal date.

CADI 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.

CADI can be used with simulation targets at any level of abstraction, such as instruction-accurate 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 Arm® Development Studio 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.

Fast Models provides some example applications that demonstrate how to use CADI in `$PVLIB_HOME/examples/CADI/`.

2.1 Benefits of CADI

A CADI implementation provides many technical benefits.

For example:

Retargetability

The interface exposes sufficient information on a target to enable describing its behavior and hardware resources. The caller does not require extra 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.

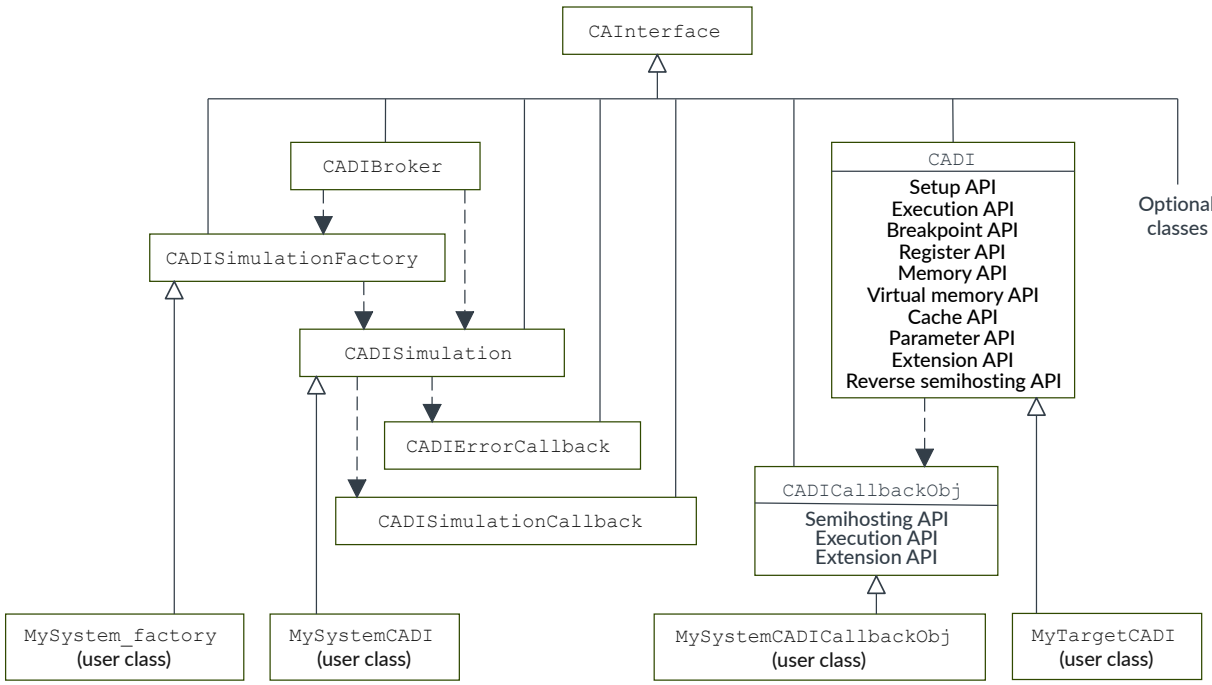
2.2 Class hierarchy

This section describes the CADI class interface.

- This guide distinguishes references to the *Component Architecture Debug Interface* (CADI) as a whole 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 2-1: CADI class hierarchy overview



Most of the CADI functionality is exposed through these classes:

`CADI`

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

The models implement `CADI` objects.

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

`CADICallbackObj`

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

The debugger must implement `CADICallbackObj` objects. Register them 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.

You could poll the state of the target model each cycle through the regular CADI interface. It is more efficient however to have the target use the `CADICallbackObj` 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 debugger can do the remaining processing 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 that are built around components and interfaces.

Simulation and factory classes

These classes provide the mechanism for creating and running simulations:

- `CADIBroker` class.
- `CADISimulationFactory` class.
- `CADIErrorCallback` class.
- `CADISimulationCallback` class.
- `CADISimulation` class.

CADI class

The methods in this class 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 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.

- `CADIDisassemblerCB` class.
- `CADIDisassembler` class.

CADI profiling classes

The profiler class enables you to record and monitor profile information about the debugging session.

- `CADIProfilingCallbacks` class.
- `CADIProfiling` class.


Note

The Fast Models processor components do not support the CADI profiling classes. This guide, therefore, contains only a high-level overview of the profiling classes.


Note

See the `CADITypes.h` file for definitions of enumerations and data structures that CADI uses.

Related information

Class reference on page 94
CAInterface class on page 94
CADIBroker class on page 96
CADISimulationFactory class on page 100
CADIErrorCallback class on page 103
CADISimulationCallback class on page 104
CADISimulation class on page 105
CADICallbackObj class on page 107
CADI class on page 112
CADIDisassemblerCB class on page 144
CADIDisassembler class on page 146
CADIProfilingCallbacks class on page 150
CADIProfiling class on page 151
Data structure reference on page 164

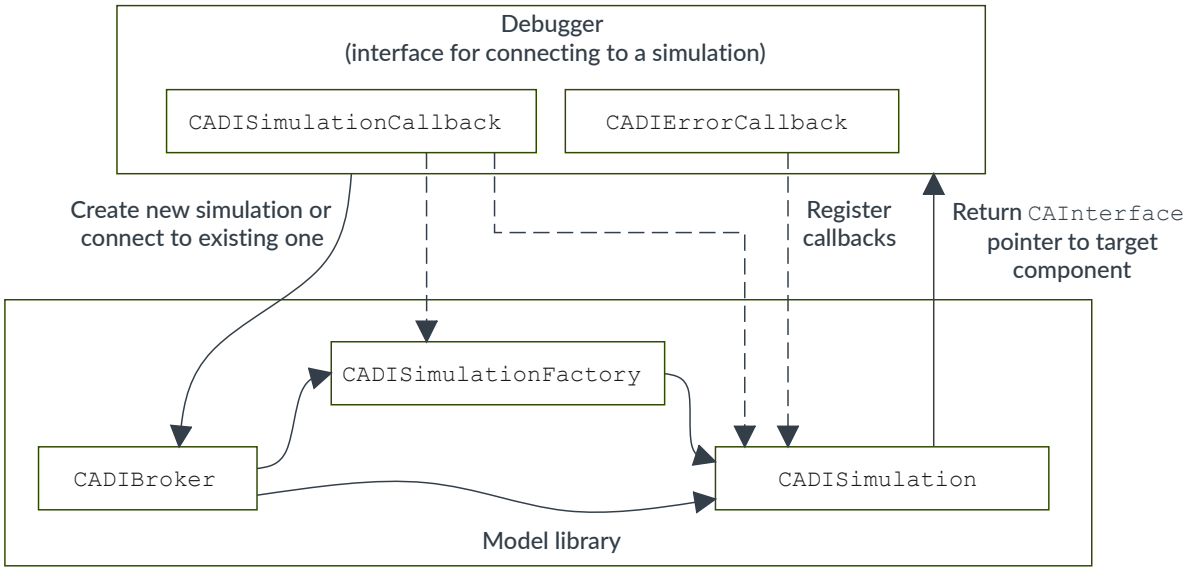
2.3 CADI classes used to connect to a simulation

This section describes the CADI classes used to connect to a simulation.

2.3.1 About the CADI classes used to connect to a simulation

This section describes the interface class, the `CADIBroker` class, `CADISimulation` class, and the `CADISimulationCallback` and `CADIErrorCallback` callback classes.

Figure 2-2: 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.

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 `CADIErrorCallback` 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.

Related information

CAInterface class on page 94

2.3.2 CADI classes used to control the simulation target

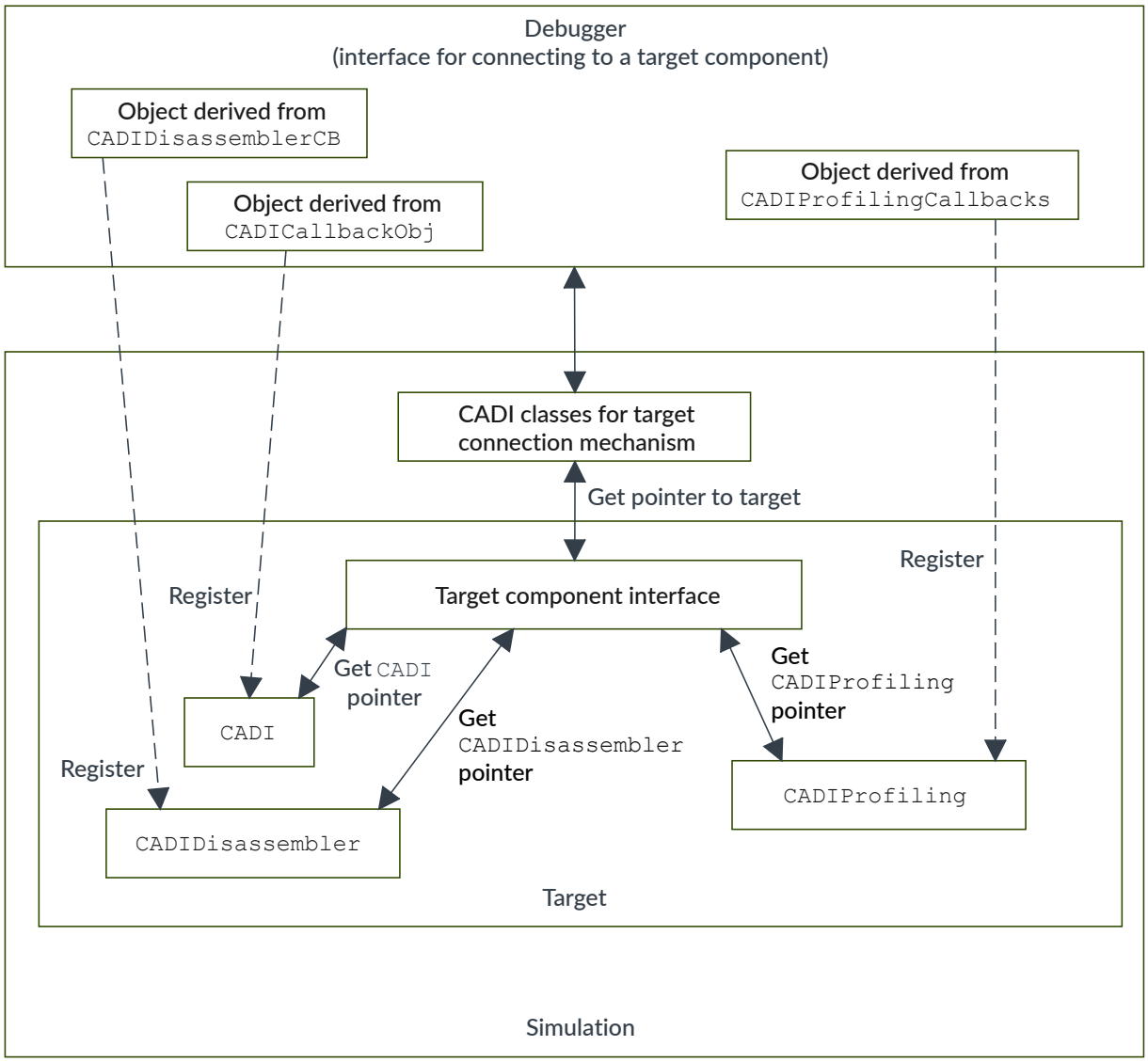
The `CADISimulation` method `GetTarget()` returns a pointer, of type `CAInterface`, to the required target component. After calling its `ObtainInterface()` method to validate interface compatibility, the target can convert the pointer to the wanted interface type.

The 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.

You can add interface extensions, alongside 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.

The following figure shows the relationship between the CADI classes used in the target and in the debugger. All of the objects shown derive from `CAInterface`:

Figure 2-3: Target interface acquisition



Related information

Optional implementation on page 24
CADI extension mechanism on page 87

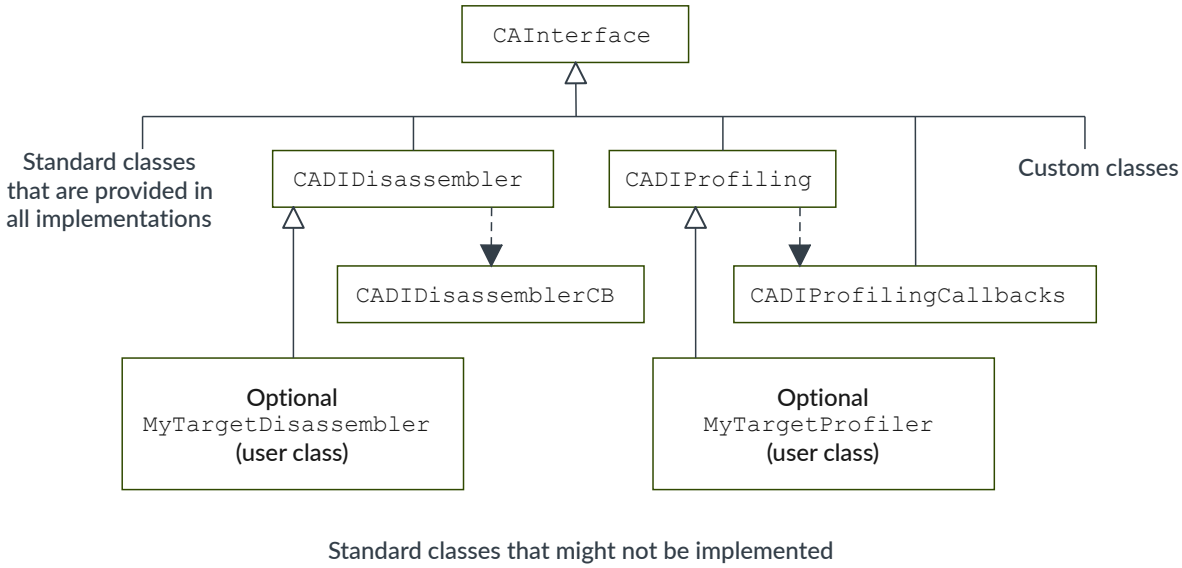
2.3.3 Optional implementation

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 are optional.

Figure 2-4: 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`.

3. Target connection mechanism

This chapter describes the target connection mechanism.

3.1 About the target connection mechanism

This section describes 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.

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 3-1: Connection sequence for existing simulation

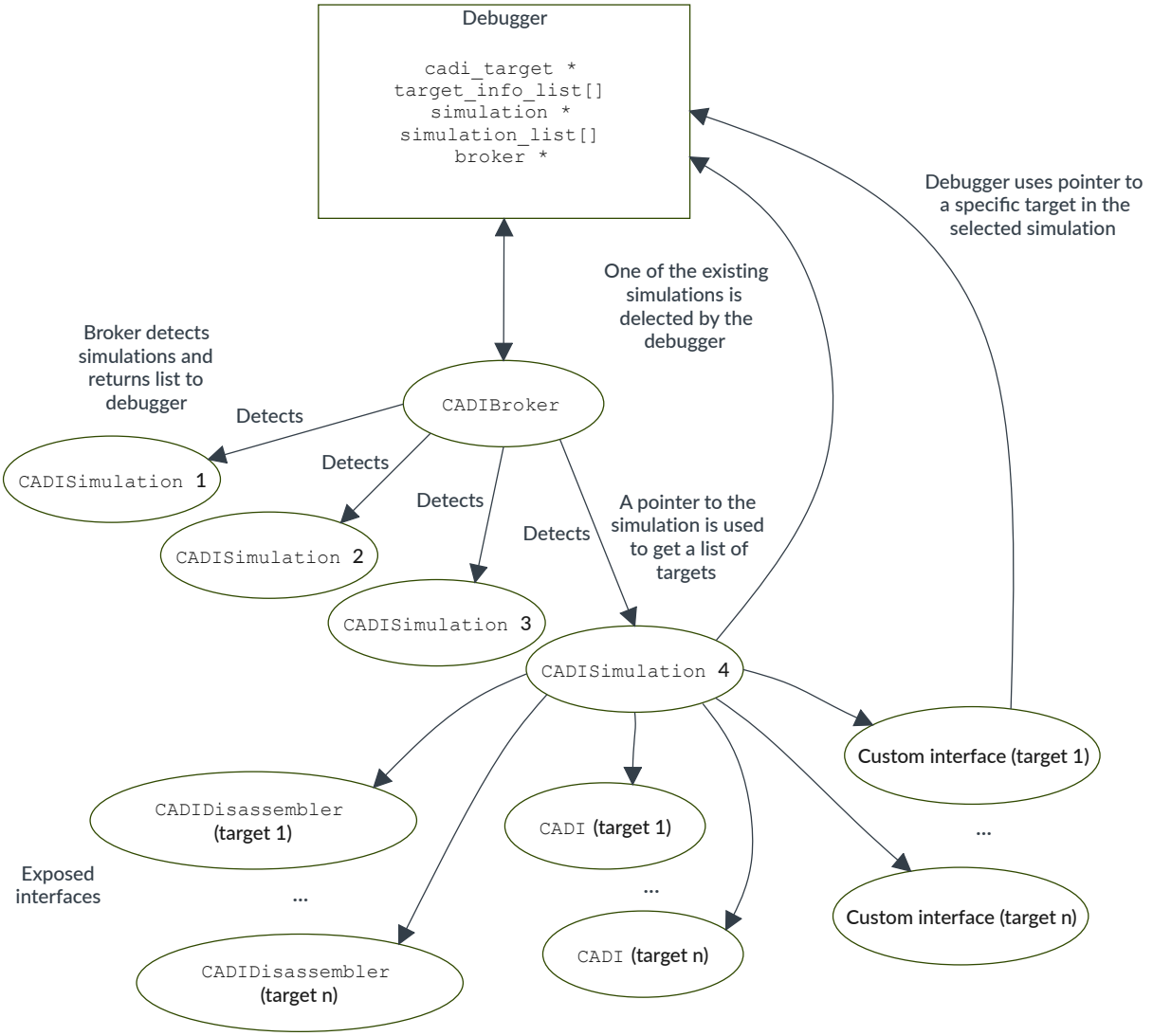
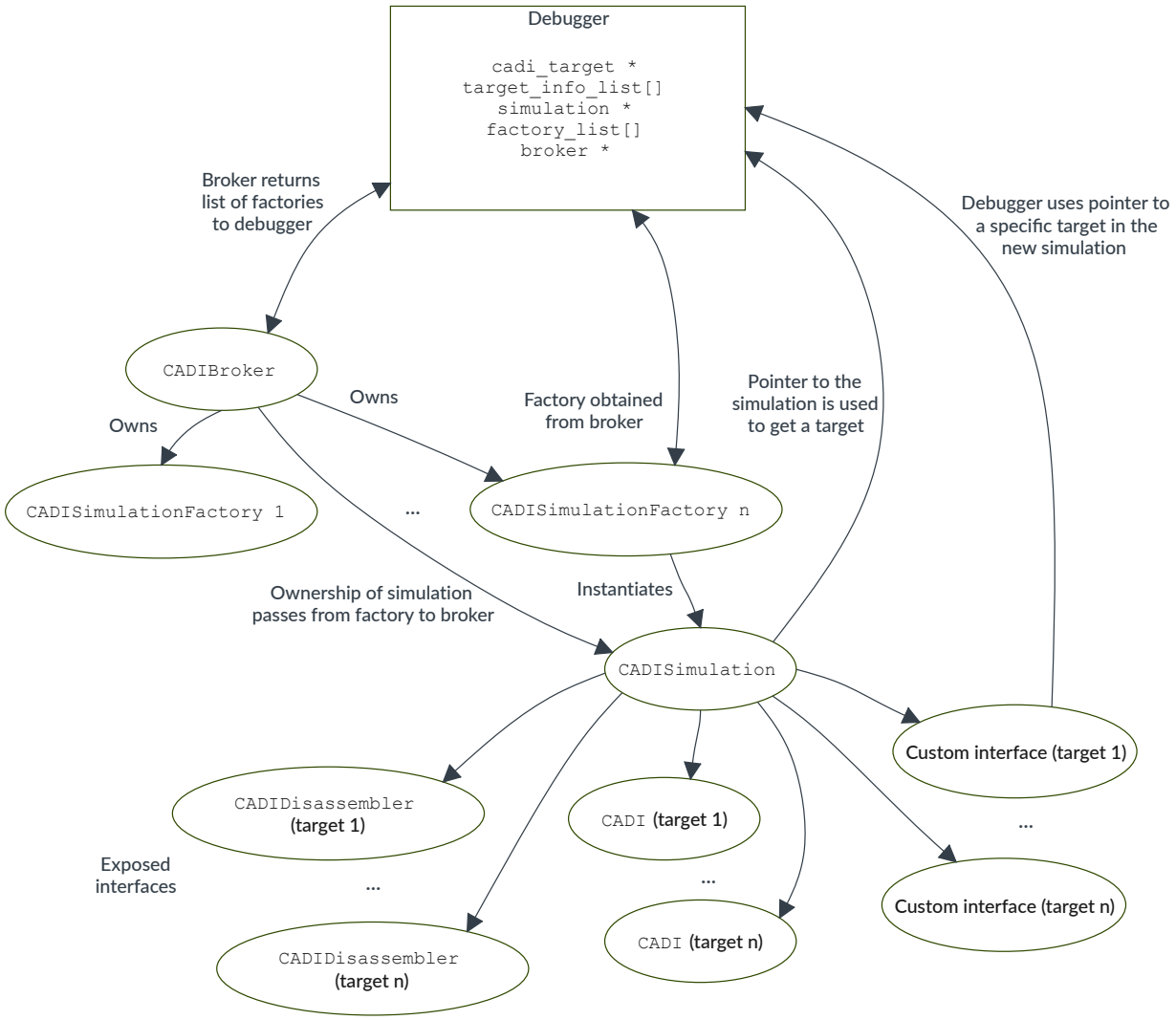


Figure 3-2: Connection sequence for new simulation



3.2 Requirements for the target connection mechanism

Implementing the target side of the CADI target connection mechanism requires one global function, and the corresponding interface classes and their methods. There are specific requirements for the implementation of each class. This section describes them.

3.2.1 CADIBroker

The `CADIBroker` is is the central element of the target connection mechanism. It establishes the connection to existing simulations and the instantiation of new simulations. This section describes it.

3.2.1.1 CADIBroker creation

The `CreateCADIBroker()` function in a model library indicates the presence of a CADI interface. The function returns a pointer to a `CADIBroker` object.

You can implement the function 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.

Example 3-1: Obtaining a pointer from a new CADIBroker object

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

3.2.1.2 CADI simulation connection

This section describes mechanisms for connecting to a simulation.

Connect 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 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.

Connect to an existing simulation: simulation selection and connection

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 the same moment that 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 a CADI broker represents the client. The server might be, for example, directly embedded into a simulation platform or implemented within an environment that runs the simulation.

Create a simulation and connect to it: simulation factory list retrieval and simulation instantiation

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 broker directly owns the simulation objects).

Related information

CADISimulationFactory on page 31

3.2.1.3 Preprocessor define settings

This section describes the preprocessor define settings.

Example 3-2: Obtaining a pointer from a new CADIBroker object

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

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.

3.2.2 CADISimulationFactory

The `CADISimulationFactory` creates a new CADI simulation.

The simulation factory:

- Provides basic information (name and a brief description) about the simulation associated with it.
- Exposes information about 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 subcomponent. 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 subsystem, or a single component, in a platform.

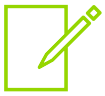
During instantiation of a CADI simulation, the corresponding interface method receives the parameters:

- 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.


Note

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.

3.2.3 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.

3.2.4 ObtainInterface()

This section describes the `ObtainInterface()` method.

`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.

Related information

Extending the target side on page 88

3.2.5 Callback objects

This section describes the callback classes that the target connection mechanism of CADI uses.

`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.

Related information

Disconnecting from a target on page 45

3.3 Connecting to a simulation

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

3.3.1 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.

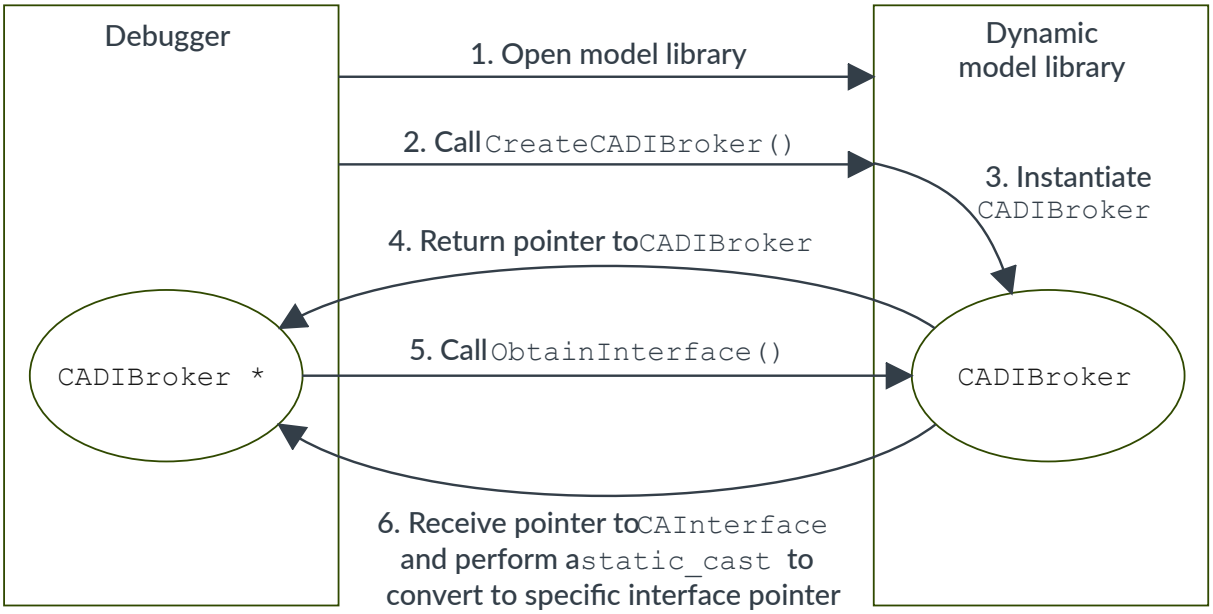
If remotely connecting to a simulation, the opened dynamic library must translate the calls arriving at the CADI interface into a format that can be transferred through a common interface such as, for example, TCP.

3.3.2 Creating the CADIBroker

After opening the library, the next step in establishing a target connection is calling `CreateCADIBroker()`. 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 3-3: 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`.

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);
}
      ...
```

Unless otherwise specified, the instructions apply to either:

- Connecting to an existing simulation.
- Instantiating a new simulation.

Alternatives for connecting to a simulation are:

- Get the simulation factories owned by the broker.
- Get the currently running simulations.

Related information

Using GetSimulationFactories() on page 36
Getting existing CADI simulations on page 40

3.4 Using GetSimulationFactories()

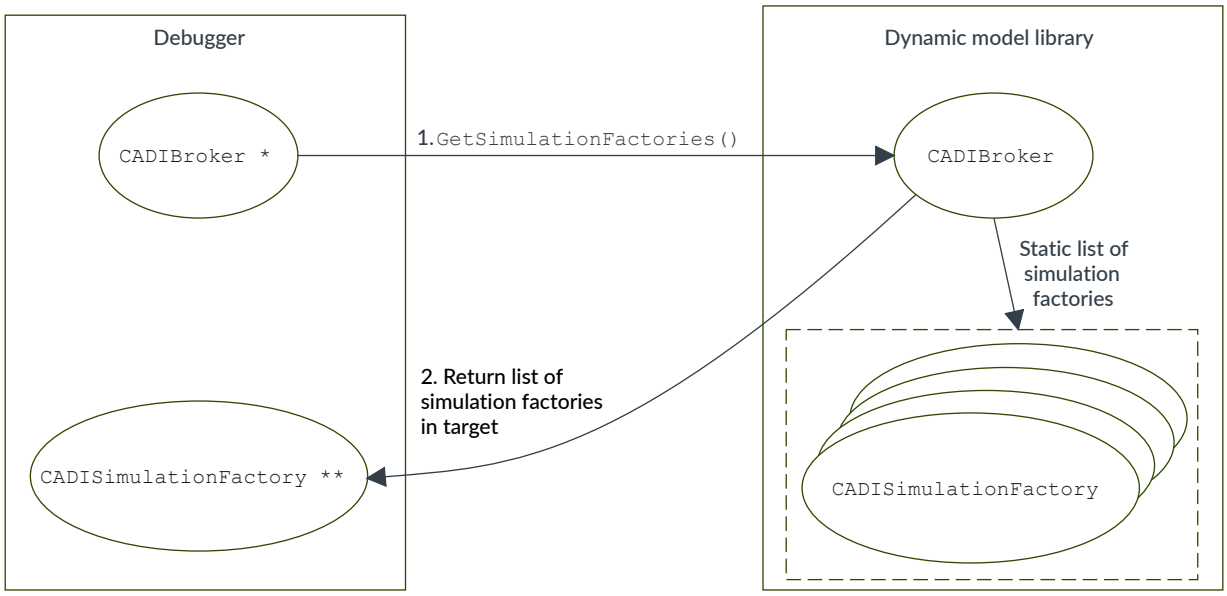
One way to establish a connection to a simulation target within CADI is to instantiate a CADI simulation and to connect to one of its targets.

To retrieve the list of available CADI simulation factories, the caller must execute the `GetSimulationFactories()` method in the CADI broker. 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.

The caller is responsible for releasing, but not deleting, all obtained simulation factory objects. It is not sufficient to release only those objects that have been used to instantiate a simulation.

Figure 3-4: 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, to create the CADI broker.

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

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

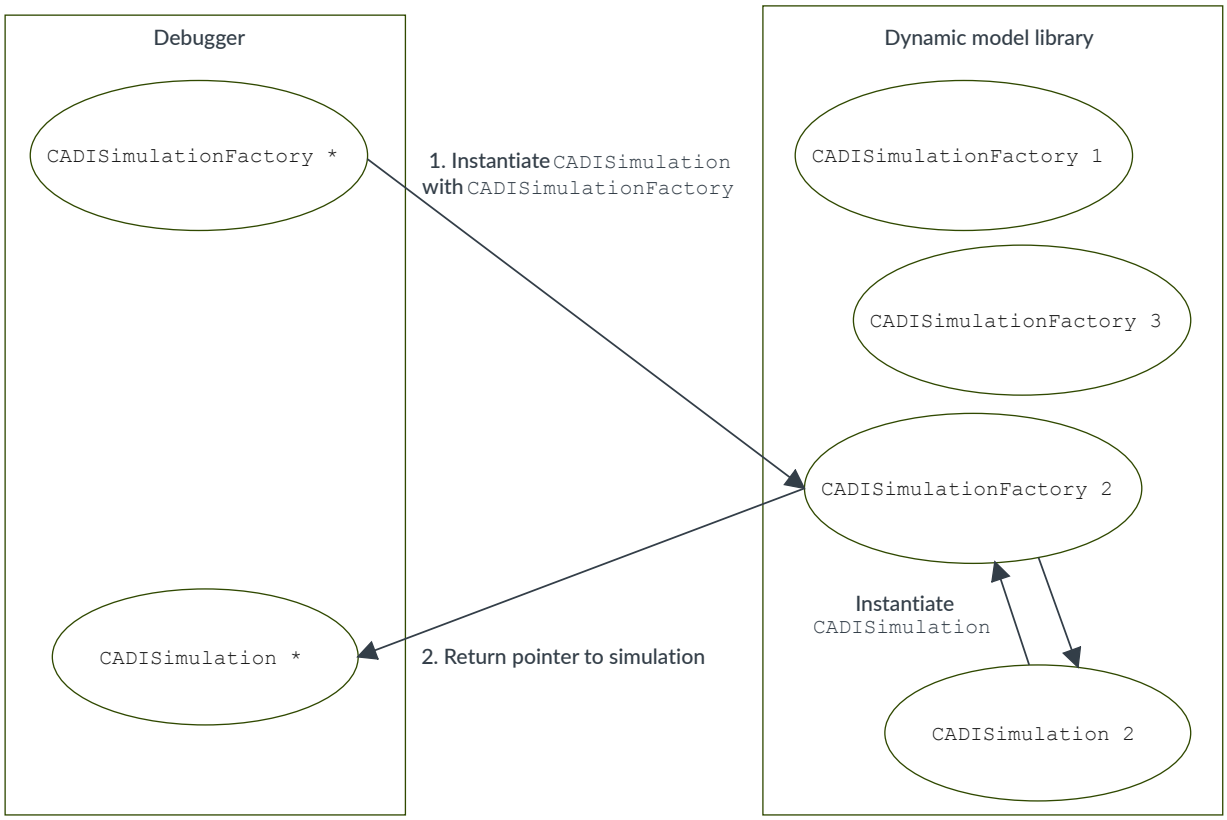
The caller can create a list of parameter values (of type `CADIParameterValue_t`) that are used for configuration of the platform. This list must end with an extra element that has the parameter ID `0xFFFFFFFF`. It is required to add this element because not all parameters require setting and the order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

Note

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

The ID `0xFFFFFFFF` is equal to `static_cast<uint32_t>(-1)`.

Figure 3-5: 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. The caller must provide them.

The result of the simulation instantiation is a pointer to a `CADISimulation` object. 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 release can be safely done for these reasons:

- The CADI broker manages the CADI simulation.
- The simulation factory can be retrieved again from the broker if necessary.

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 chosen, 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...
```

Related information

Creating the CADIBroker on page 35

3.5 Getting existing CADI simulations

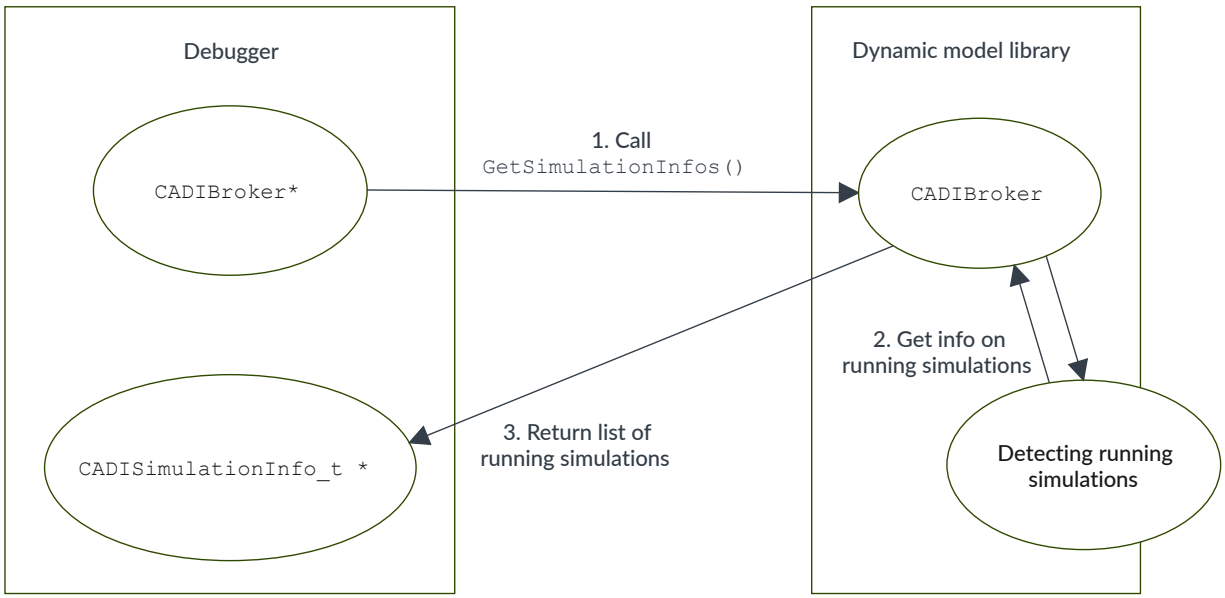
If the caller connects to a running CADI simulation, it must retrieve information on this simulation by calling the `GetSimulationInfos()` method of the CADI broker. This call returns an internal list of available simulations that the broker maintains.

The number of elements that are retrieved depends on:

- The size of the buffer that is 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 simulations that the broker holds can change dynamically. Consider updating this list regularly to detect the creation or destruction of CADI simulations.

Figure 3-6: Getting information on existing CADI simulations

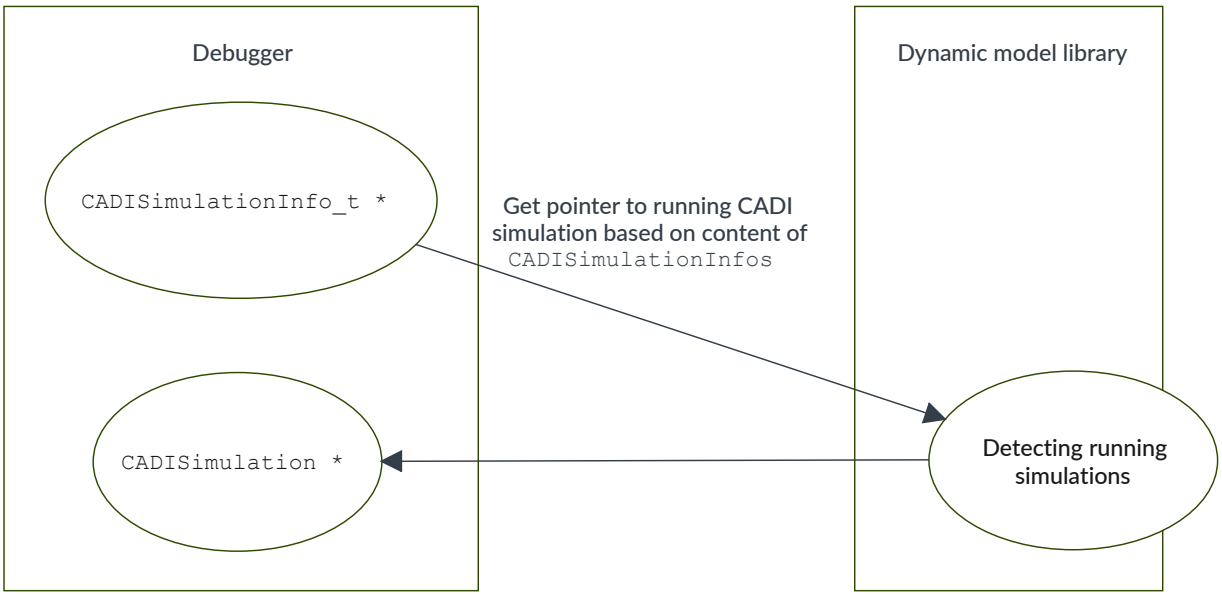


Based on the acquired information, the caller uses `SelectSimulation()` to select a simulation to attach to. To specify a simulation, its ID (as part of the simulation info) must be used.

`SelectSimulation()` can receive pointers to a `CADIErrorCallback` object and a `CADISimulationCallback` object. These objects are automatically registered to the requested simulation. The caller must provide these objects.

It is not required that a specific simulation ID matches the corresponding index for the simulation within the internal list that the broker holds.

Figure 3-7: 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.

A typical implementation for 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)
```

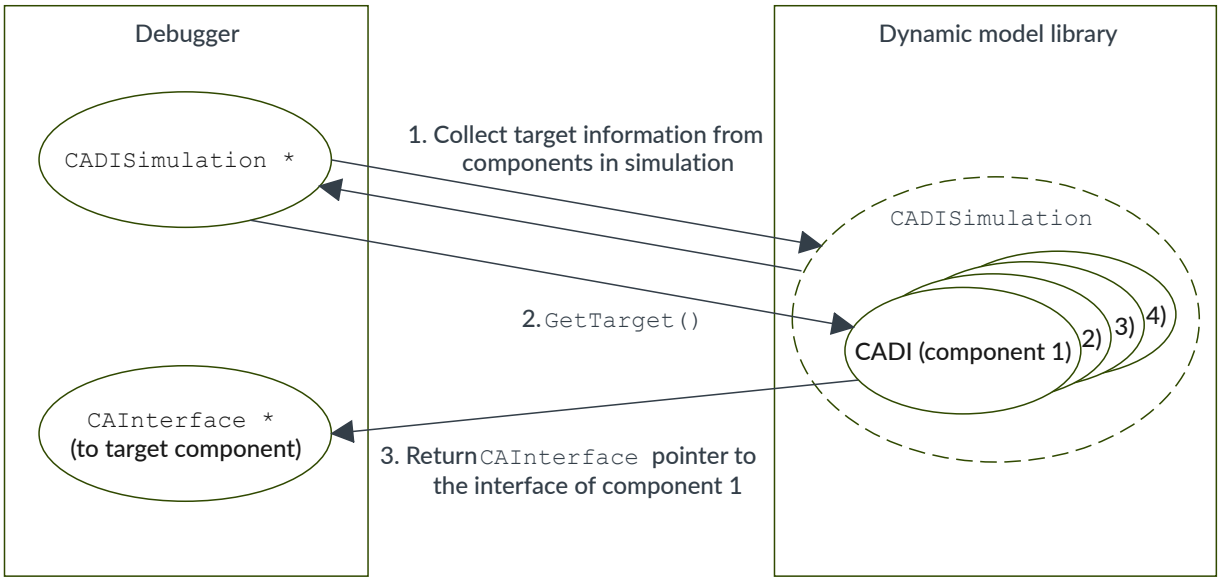

Note

The size of `eslapi::CADISimulationInfo_t` is 8kB. When allocating arrays of this object on the stack, do not to exceed the stack allocation limits.

3.6 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 3-8: Getting a target interface



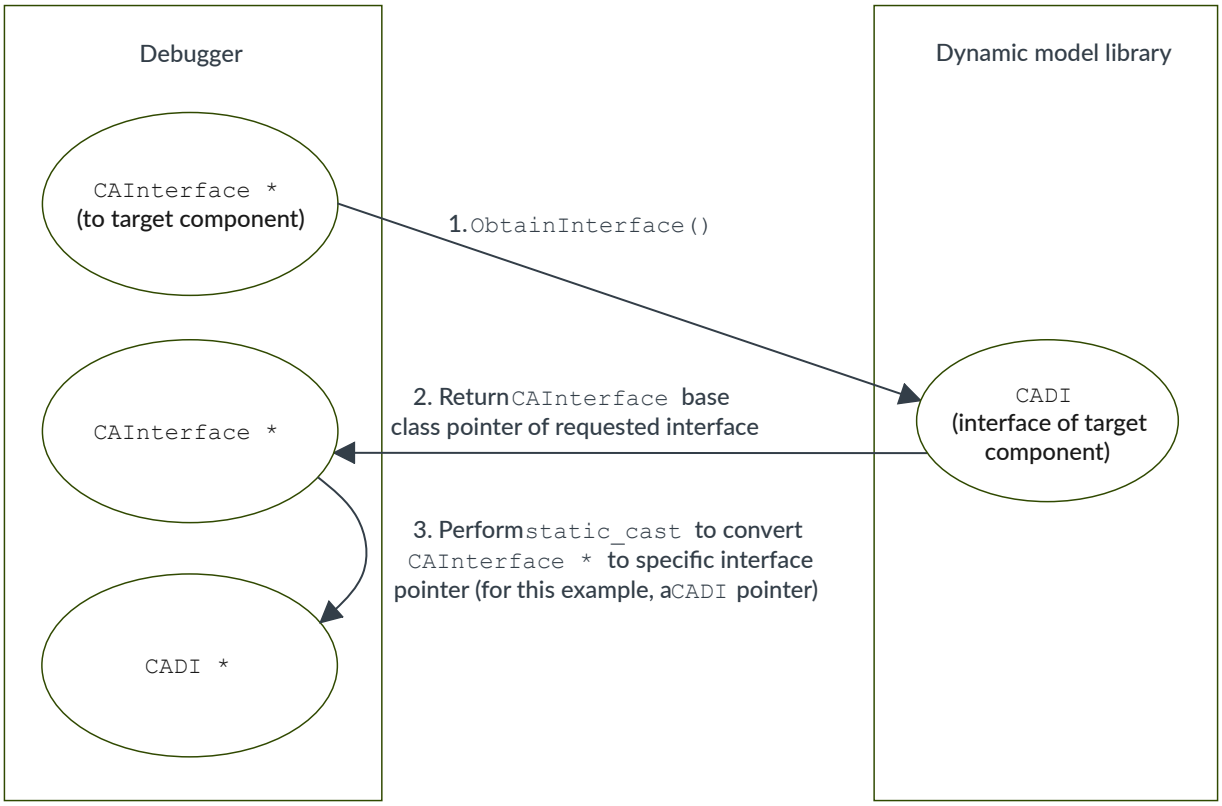
The `CADISimulation` class holds information on the contained target components that can be retrieved using the `GetTargetInfos()` method. This information includes the ID and properties of the target that might be important for a debugger such as, for example, whether the target executes software.

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`, 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 3-9: Getting a CADI interface



Typical implementation for 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 ...
```

3.7 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.

3.7.1 About disconnecting from a target

This section describes how to disconnect from a target.

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.

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.

3.7.2 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.

3.7.3 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 3.4 Using GetSimulationFactories() on page 36, 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.

Related information

Using GetSimulationFactories() on page 36

3.7.4 Typical shutdown scenarios

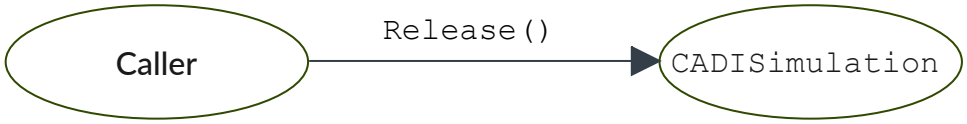
This section describes the typical scenarios for shutting down a simulation.

3.7.4.1 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 value. Depending on the parameter value, the simulation is destroyed or kept alive.

Figure 3-10: Single caller and simulation shutdown initiated by caller



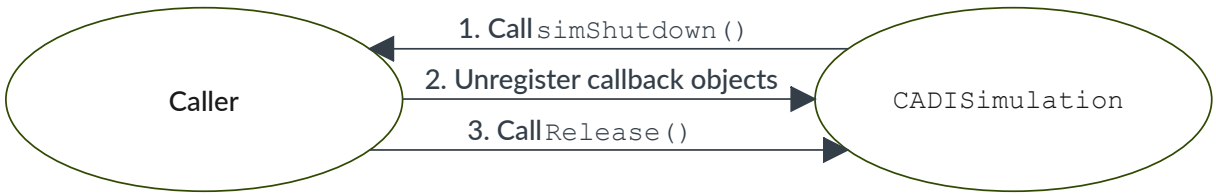
3.7.4.2 Single caller and the simulation initiates shutdown

The simulation initiates its shutdown and informs the caller.

This scenario is used, for example, if the simulation offers a user interface for interaction that permits ending the simulation. The procedure requires these 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.
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 will not access the simulation or targets in the future.

Figure 3-11: Single caller and simulation shutdown initiated by simulation

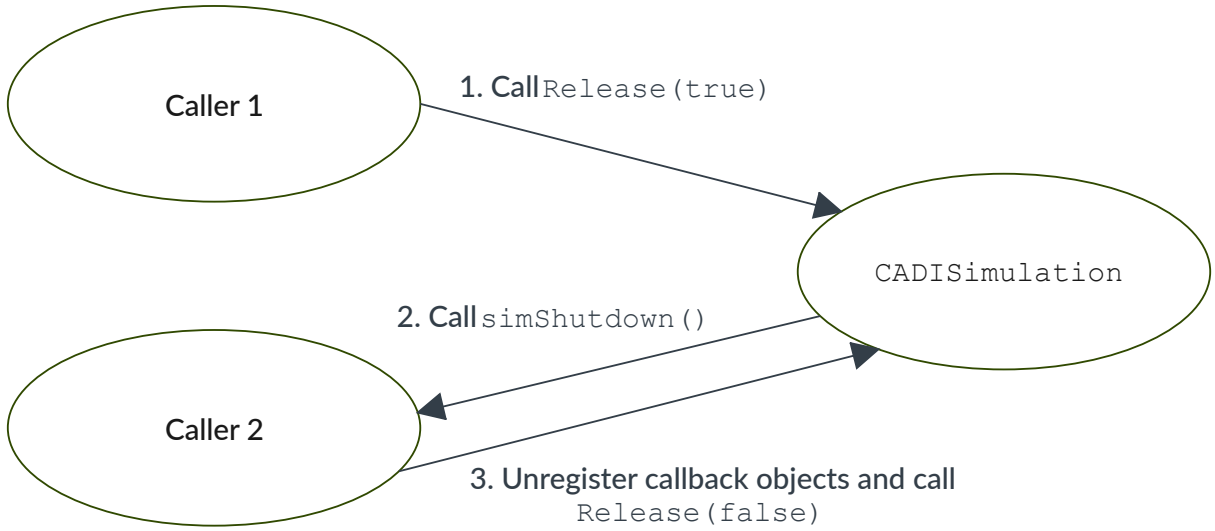


The shutdown parameter can be set to `false`, as the simulation is already shutting down. A value of `true` is ignored at this point.

3.7.4.3 Multiple callers and one of the callers initiates shutdown

The sequence is similar to that for a single caller that initiates shutdown except that the other caller must also be shut down.

Figure 3-12: Simulation shutdown initiated by caller while multiple callers are attached



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.

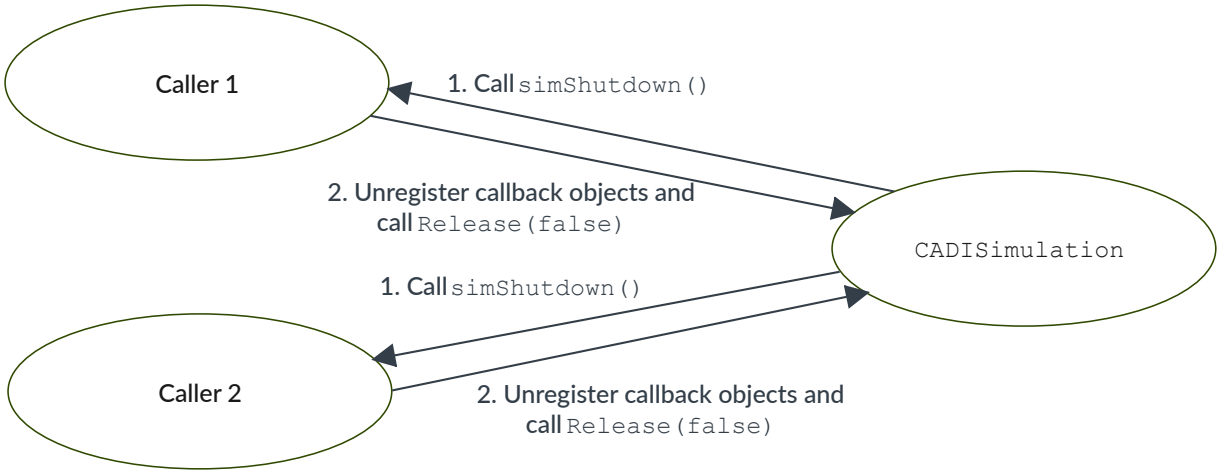
   The affected callers must sign off with a `Release()` call to announce successful disconnection from the simulation. Its `shutdown` parameter is set to `false` as the shutdown is already in progress (a value of `true` is ignored at this point).
4. After all callers have disconnected from the simulation,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

3.7.4.4 Multiple callers and the simulation initiates shutdown

Multiple callers are attached to a simulation and the simulation initiates its own shutdown.

This scenario is used, for example, if the simulation offers a user-interface for interaction that permits ending the simulation:

Figure 3-13: Multiple callers and simulation shutdown initiated by simulation



The main difference between this situation and one in which there are multiple callers and one of the callers initiates shutdown 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.

4. Using the CADI interface methods from a debugger

This chapter describes how a debugger uses the CADI interface to control the target.

4.1 CADI accesses from a debugger

This section describes CADI accesses from a debugger.

4.1.1 About 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.

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 this form:

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

Follow these guidelines 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.

4.1.2 CADI and threads

Debugging a simulation model that exposes a CADI implementation typically uses one simulation thread and 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 these 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.

4.2 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.

4.3 Target connection and configuration

This section summarizes the steps for target connection and configuration.

4.3.1 Connecting to targets

This section describes conceptually how to connect to targets.

About this task

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

Procedure

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.

Related information

Target connection mechanism on page 26

4.3.2 Obtaining an interface pointer to the target

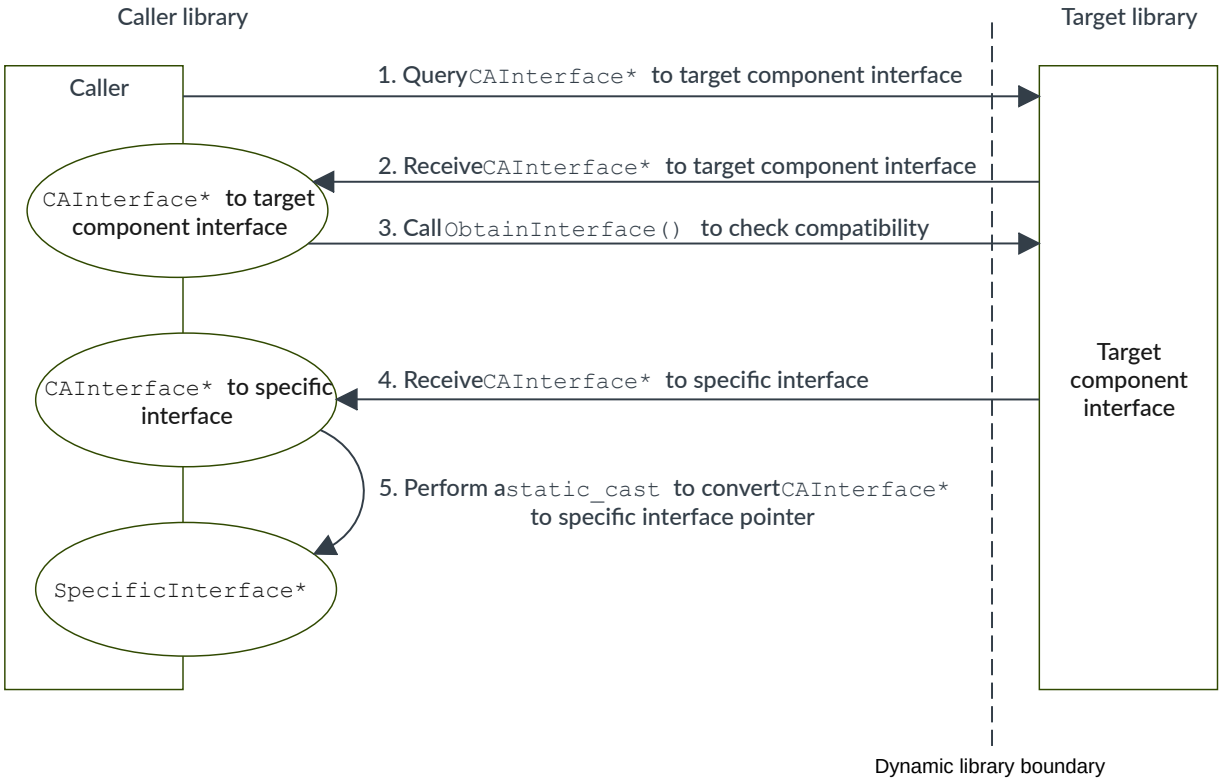
This section describes the steps to obtain the interface pointer.

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 4-1: Obtaining a pointer to a specific interface



4.3.3 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.

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.

Related information

CADI target characteristics on page 57

4.3.4 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.

4.3.5 CADI target characteristics

This section describes CADI target characteristics.

4.3.5.1 About 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.

4.3.5.2 Extended Target Features Register

This is an important target feature for helping a debugger to adjust to the current target. 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:

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

Arm recommends adding a colon at the end of the string, as shown.

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.

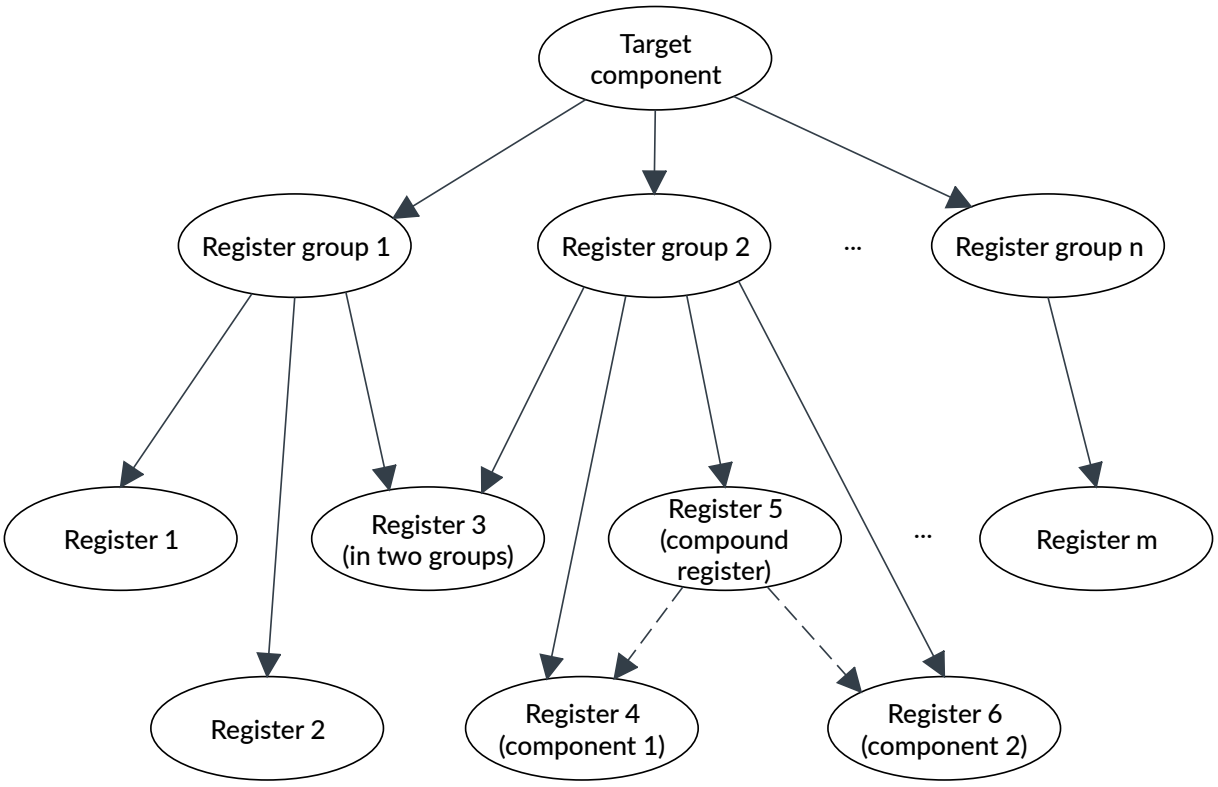
Related information

CADITargetFeatures_t on page 168

4.3.6 Querying the hardware resource for register information

The register information is organized hierarchically. The caller must query this hierarchy to obtain information on a specific register.

Figure 4-2: Register organization



The first step is to examine the information that the target features provide. It contains the number of available register groups. Calling `CADIRegGetGroups()` for a specific group retrieves more detailed information. The call scheme is similar to a typical one.

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 contrast to a typical call scheme, this method additionally 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.

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. The method must result in an array that is larger than (if there are multiple entries) or equal to the required size:

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. Compound registers are composed of several other registers. For example, a 32-bit integer register might be 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. You can 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`.

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.

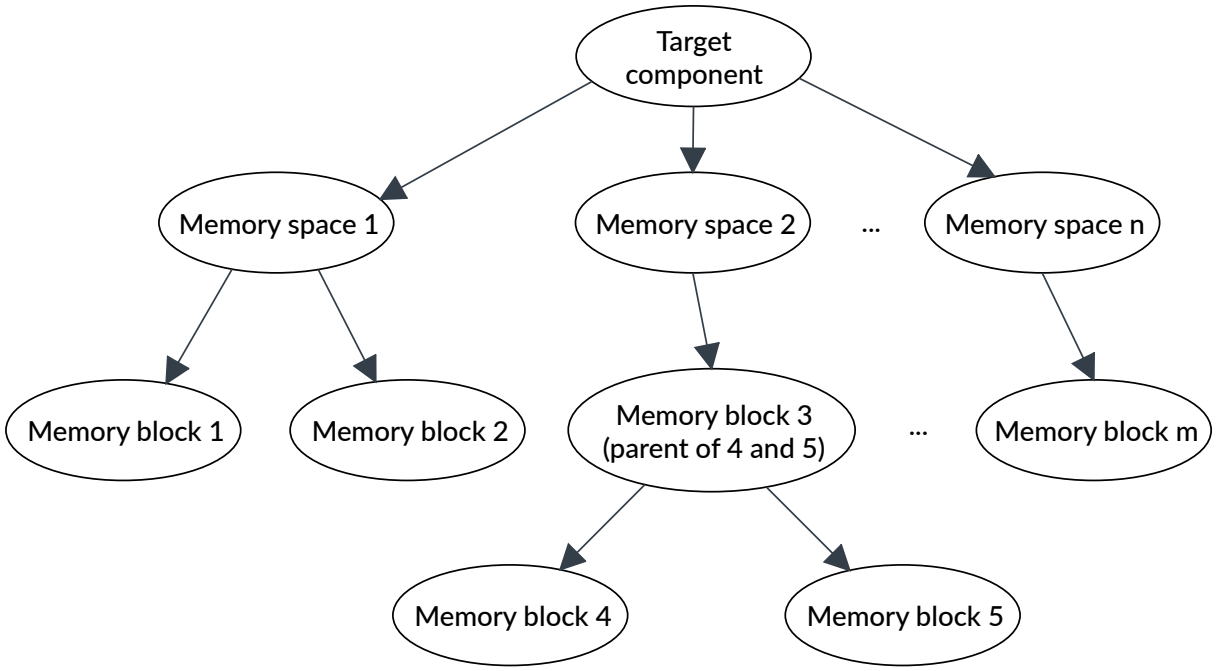
Related information

CADI accesses from a debugger on page 50

4.3.7 Querying the hardware resource for memory information

Similar to register information, memory information has a hierarchical structure.

Figure 4-3: 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 used for CADI accesses from a debugger.

A memory space is subdivided into memory blocks that define the characteristics of certain ranges of memory within a memory space, such as 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. 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`.

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;
// ...
```

Related information

CADI accesses from a debugger on page 50

4.4 Register access

This section describes how to access registers in the target.

4.4.1 About accessing registers

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

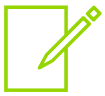
The elements of the array:

- Specify the addressed register by its register number (ID).
- Provide a buffer of 16 bytes for accesses.
- Receive information about permitted access (read, write or read-write).
- Optionally specify 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.
- Enable the target to indicate registers with undefined content.

Accessing registers in the target

```
// One way to implement a read access to a register with a width of 512 bits.
// "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.



The contents of the `CADIReg_t` data buffer must be accessed in little endian, even if the target uses a different endianness. 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`.

4.4.2 Reading from string registers

Reading from string registers works slightly differently to reading from an integer or a floating-point register. In contrast to other types of registers, a string register does not own a bitwidth.

The string itself determines the actual size of the string that is read through the string register. 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), increase the `offset128` parameter and read the register after every set of 16 bytes.

Reading 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
   }
}
```

4.4.3 Writing to string registers

Writing to string registers works differently to writing to an integer or a floating-point register, and to reading a string register. In contrast to other types of registers, a string register does not have a fixed bitwidth.

A `CADIRegWrite` to a string register using nonzero `offset128` could extend, truncate, or update a string. To avoid ambiguity, string updates must allocate an array of n `CADIReg_t` elements with enough buffer space to store the entire string, including a terminating null character. The `offset128` parameter in each `CADIReg_t` must have the value *n* and the bytes buffer must contain the *n*th 16 byte chunk of the string. The caller performs a single `CADIRegWrite`, updating the string register atomically, if successful.

Writing string registers

```
eslapi::CADIReturn_t status;
// "register_info" contains information on a string register
std::string writeString("Pneumonoultramicroscopicsilicovolcanoconiosis");
const char *s = writeString.c_str();
uint32_t bytes = strlen(s) + 1;
uint32_t chunks = (bytes + 15) / 16; // The number of 128-bit chunks required to hold the null
 terminated string
uint32_t numRegsWritten = 0;
eslapi::CADIReg_t *regs = new eslapi::CADIReg_t[chunks];
for(uint32_t i = 0; i < chunks; i++)
{
   regs[i].regNumber = register_info.regNumber;
   regs[i].offset128 = i;
   uint32_t remaining = bytes - i*16;
   memset(regs[i].bytes, 0, 16);
   memcpy(regs[i].bytes, &s[i*16], remaining > 16 ? 16 : remaining);
}
status = cadi->CADIRegWrite(chunks, regs, &numRegsWritten, 0);
delete[] regs;
if (status != eslapi::CADI_STATUS_OK || numRegsWritten != chunks)
{
   printf("ERROR: Writing register failed\n");
   return;
}
```

4.5 Memory access

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 endianness.

The total memory accessed in bytes is equal to the number of access units multiplied by their size in bytes. The `data` buffer that is 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` 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 number of completed units, and the contents of `data` is valid for additional processing. An example of such a situation is an attempt to access memory that is not part of a memory block. This might happen when performing an access that exceeds a valid memory range.

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 cannot interfere with the execution of the target.

A side effect during writing to memory might be for example the usage of a memory-mapped register whose contents control 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.

Writing to memory

```
eslapi::CADI* cadi;
eslapi::CADIMemSpaceInfo_t mem_space;
eslapi::CADIMemBlockInfo_t mem_block;
// ...fill the variables declared above 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 4-byte words.
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;
```

4.6 Execution control

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.

4.6.1 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.

4.6.1.1 Predefined breakpoint types

CADI provides 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

Do not confuse these enum data types:

- `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 enum data types, `TypeExtension` is one of these:

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

4.6.1.2 Breakpoint properties

`CADIBptRequest_t` owns several fields specific to certain breakpoint types. These fields are ignored for other types.

This sections gives an overview of the respective associations between fields in `CADIBptRequest_t` and the various breakpoint types.

Table 4-1: 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

triggerType	Program	Memory	Register	Instruction Step	Program Range	Exception
Conditions	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]	Yes[1]
useFormalConditions	Yes	Yes	Yes	Yes	Yes	Yes
formalCondition	Yes[2]	Yes[2]	Yes[2]	Yes[2]	Yes[2]	Yes[2]
type	Yes	Yes	Yes	Yes	Yes	Yes
regNumber	-	-	Yes	-	-	-
temporary	Yes	Yes	Yes	Yes	Yes	Yes
continueExecution	Yes	Yes	Yes	Yes	Yes	Yes

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`.

4.6.1.3 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.

These details include:

- 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.

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

[1] Yes only if `useFormalConditions` is 0.
[2] Yes only if `useFormalConditions` is 1.

is a signed or unsigned value, and by the bitwidth specified in the condition data type. The bitwidth includes the sign bit.

Related information

Breakpoint properties on page 69

4.6.1.4 Breakpoint management

To set a new breakpoint, call `CADIBptSet()`. It receives a breakpoint description of type `CADIBptRequest_t`. On return, the caller receives a breakpoint ID of type `CADIBptNumber_t` to use in subsequent breakpoint management calls.

`CADIBptSet()` only supports setting breakpoints in virtual memory spaces. Setting a breakpoint in a physical memory space returns `CADI_STATUS_ArgNotSupported`.

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.

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 that used by CADI accesses from a debugger. 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.

Related information

Breakpoint configuration on page 70
CADI accesses from a debugger on page 50

4.6.2 Execution mode control

This section describes how to control the execution mode.

4.6.2.1 About execution mode control

To provide fully controlled debugging of the target, the attached debugger must be able to control the execution of the target.

CADI provides this capability with 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.

Arm does not recommend polling of the target state, however. 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

You cannot return `CADI_EXECMODE_Bpt` 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

You cannot use `CADIExecSetMode()` to set the target state to `CADI_EXECMODE_Bpt`.

The following code example shows how to modify the target execution mode:

```
// 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;
```

4.6.2.2 Starting and stopping the target

For a subset of execution modes, the 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, 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.

4.6.2.3 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.

4.6.2.4 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.


Note

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 this behavior in 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).

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.

Related information

Application loading on page 77

4.6.2.5 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 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 a future version of CADI will deprecate the `reset()` callback.

Related information

CADICallbackObj::modeChange() on page 110

4.6.2.6 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 initiate 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 execution control. These include:

- Semihosting.
- 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 multiple debuggers to connect 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 them 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 in 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`
- `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 of mode 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 of mode 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 change of mode 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 change of mode 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 it was responsible for the reset, to control the execution mode. The caller might expect characteristic sequences of `modeChange()` callbacks in response to a specific requested functionality.

Table 4-2: 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)`

Related information

Using the semihosting API on page 83
Execution mode control on page 72

4.7 Application loading

The CADI interface simplifies the loading of an application from a debugger to a target.

A debugger typically writes the application program code directly to the platform memory. For simplicity, CADI has a `CADIExecLoadApplication()` method that autonomously writes the application code to the target. The debugger must extract debug information, if available, from the executable. You can use this debug information to initialize more hardware resources of the simulation model: for example, by 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.

The types of executable that a model supports depends on the implementation. For example, ELF file support.

You can load multiple applications to a target, for example to load several different applications to a cluster. 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. Other debuggers connecting to this processor can use this data to obtain the required debug information.

Preserve the list of loaded applications 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 method removes the application information and any 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

Debug information support depends on the implementation of the model. This support is not necessary because the debugger side must extract the information from the application image.

4.8 CADI Disassembler

This section describes the CADI Disassembler.

4.8.1 About the 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.

4.8.2 Obtaining a CADI Disassembler

A pointer to a certain `CADIDisassembler` object is obtained from the corresponding `CAInterface` instance in the target.

The `CADIGetDisassembler()` method of the corresponding `CADI` object is retained only for compatibility with old CADI versions. Arm deprecates it. Do not use it in new implementations.

Related information

CADI classes used to control the simulation target on page 23

4.8.3 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.

Table 4-3: 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.

4.8.4 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 for the instruction set and the current mode for the processing unit.


Note

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.

4.8.5 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 enum type informs the debugger about more details of an uncompleted method call. Methods that use this return type are those that request disassembly information. These 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.

4.8.6 Disassembly acquisition

Call `GetDisassembly()` to get the disassembly from a CADI Disassembler.

The `GetDisassembly()` method has these 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

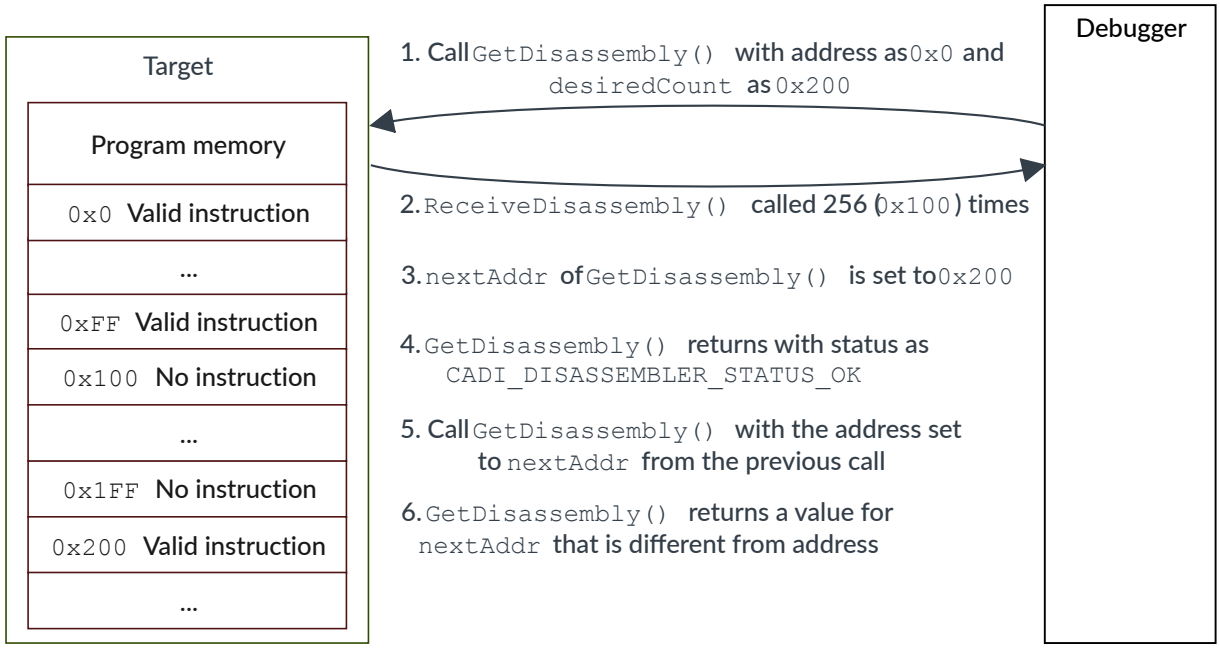
The `desiredCount` refers to the number of requested instructions. If the mode ID is 0, the size of the instruction words can vary if the mode changes in between. It is therefore possible that the distance between the addresses (as returned by the callback) is not equally spaced.

It might be necessary to update `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 these 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.

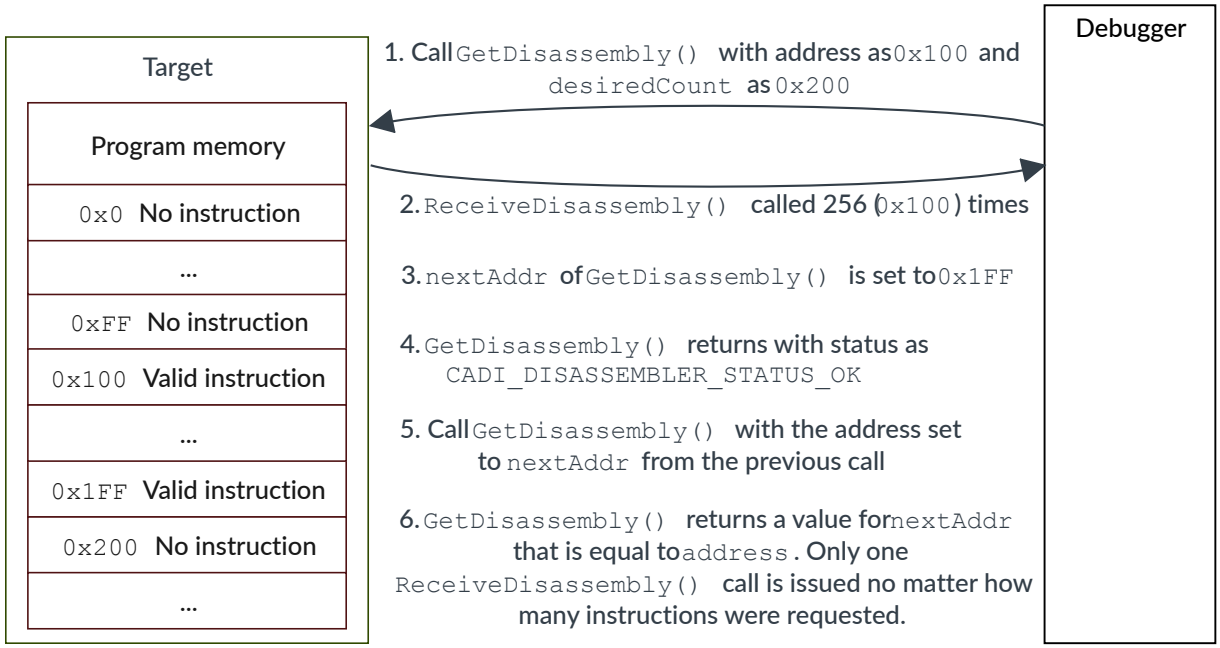
The following figure shows a call where the last instruction in the range is a valid instruction:

Figure 4-4: nextAddr set to last instruction



The following figure shows a call where the last instruction in the range is the last instruction in the memory space:

Figure 4-5: nextAddr set to last valid instruction

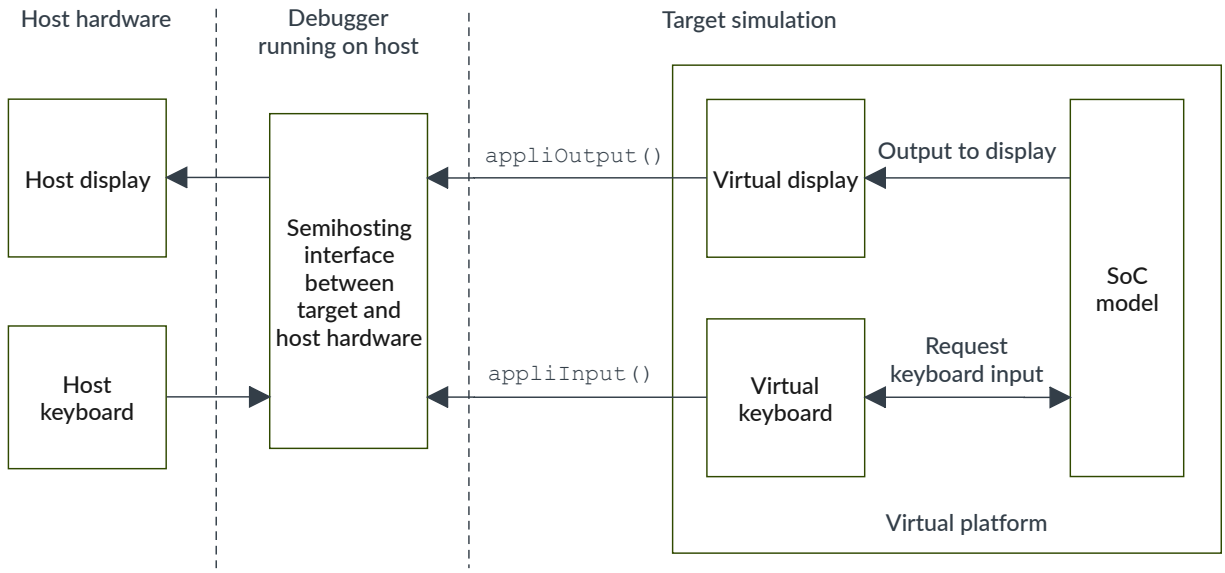


4.9 Using the semihosting API

CADI provides a semihosting interface that enables interaction between a user and a connected target.

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 4-6: 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()`. 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 enum 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.

Related information

CsADICallbackObj::appliInput() on page 108
CADICallbackObj::appliOutput() on page 109
CADICallbackObj::doString() on page 109

4.10 Profiling

These methods give access to execution and memory debug-profiling for a processor.

- Fast Models does not implement the CADIProfiling class. It is not, therefore, covered in detail here.
- 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).

Related information

CADIProfilingCallbacks::profileResourceAccess() on page 151
CADIProfilingCallbacks::profileRegisterHazard() on page 151
CADIProfiling::CADIProfileSetup() on page 153
CADIProfiling::CADIProfileControl() on page 153
CADIProfiling::CADIProfileTraceControl() on page 154
CADIProfiling::CADIProfileGetExecution() on page 154
CADIProfiling::CADIProfileGetMemory() on page 155
CADIProfiling::CADIProfileGetTrace() on page 156
CADIProfiling::CADIProfileGetRegAccesses() on page 157
CADIProfiling::CADIProfileSetRegAccesses() on page 157
CADIProfiling::CADIProfileGetMemAccesses() on page 158
CADIProfiling::CADIProfileSetMemAccesses() on page 159
CADIProfiling::CADIProfileGetAddrExecutionFrequency() on page 159
CADIProfiling::CADIProfileSetAddrExecutionFrequency() on page 160
CADIProfiling::CADIGetNumberOfInstructions() on page 160
CADIProfiling::CADIProfileInitInstructionResultArray() on page 161
CADIProfiling::CADIProfileGetInstructionExecutionFrequency() on page 161
CADIProfiling::CADIProfileSetInstructionExecutionFrequency() on page 162

5. CADI extension mechanism

This chapter describes the CADI extension mechanism that adds interfaces to a target and the modifications that are required on both the caller side and the target side.

5.1 Overview of the extension mechanism

A major feature introduced with CADI 2.0 is the extension mechanism.

The extension mechanism:

- 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 must consider:

1. Declaring a class with the custom interface extensions that must be derived from `CAInterface`. The inherited method calls must be implemented.
2. Implementing `ObtainInterface()` for the custom extension so that all existing interfaces are accessible.
3. Linking the extension to other implemented interfaces provided by the target through their `ObtainInterface()` implementations.

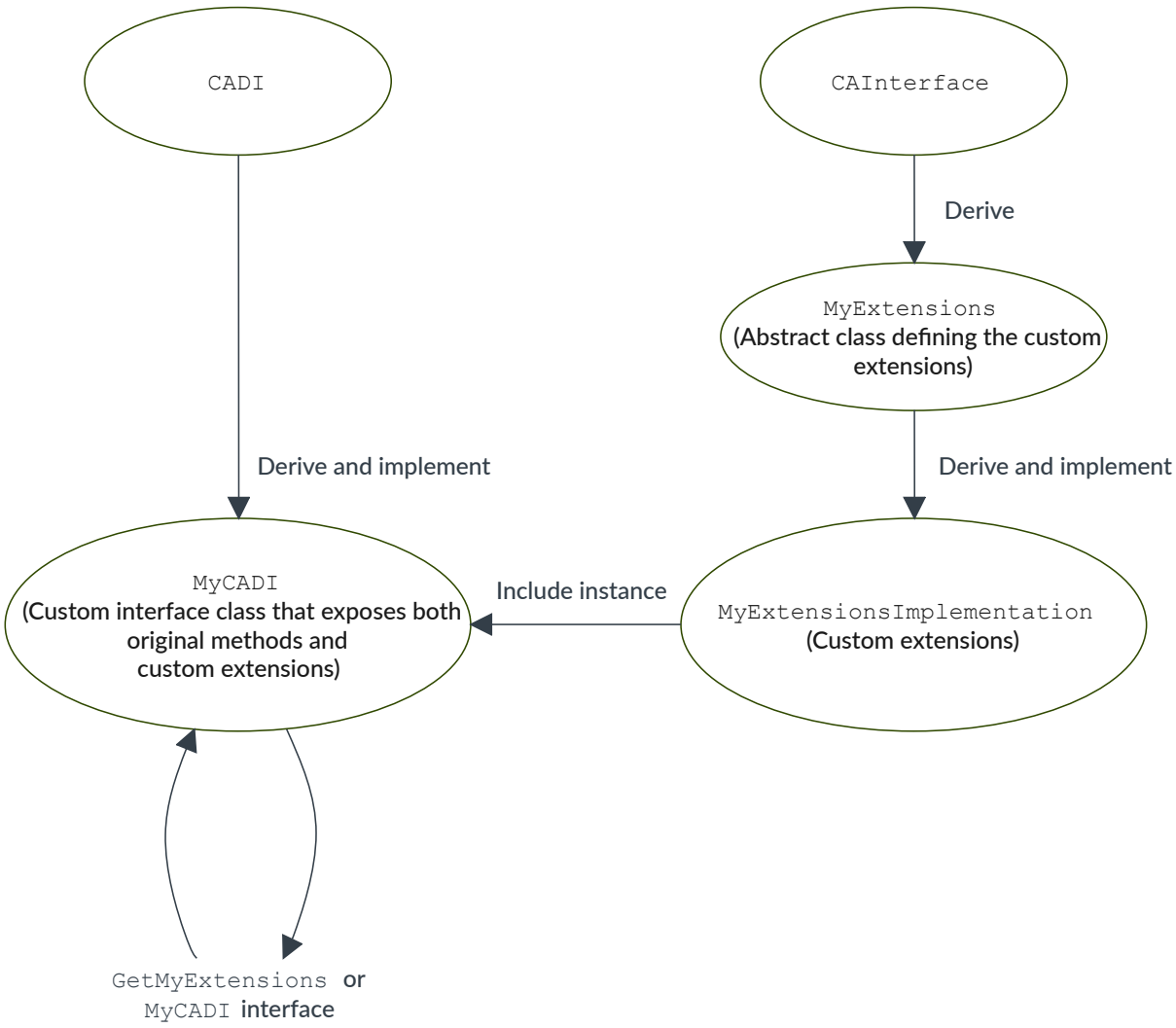
Related information

Extending the target side on page 88

5.2 Extending the target side

This section describes a way to create a simple extension interface, and the required steps to use the extension mechanism in an implementation.

Figure 5-1: Custom extensions to a CADI interface class, showing the class relationships



To create the target-side implementation:

1. Declare the interface that provides the custom extensions in a new class, called for example `MyExtensions`.

   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 the example below, the passing of pointers from the instantiated interface objects.

   The example below 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 the example above.

   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 the example below. 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 these 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.

   ObtainInterface(), a typical implementation for the extension

```
// 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.

   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 the example above, 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 above is the implementation of `ObtainInterface()` in the code fragment listed below:

Using ObtainInterface()

```
// 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;
}
```

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`.

5.3 Obtaining a custom interface

This section describes how to ensure the correct functionality of an 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 for the standard interfaces:

1. A `CAInterface` pointer to the target interface class is required. The CADI simulation typically returns this pointer.
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()`.

The following code example shows how to use `CADISimulation` to return a pointer to the interface, using the `MyExtensions` class implementation:

```
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
```

Related information

Extending the target side on page 88
Obtaining an interface pointer to the target on page 54

Appendix A Class reference

This appendix describes the classes that create, initialize, and communicate with a simulation.


Note

Implementing the `CADIDisassemblerCB`, `CADIDisassembler`, `CADIProfilingCallbacks`, and `CADIProfiling` classes and the methods that use them is optional. Typically, only components that execute applications use them.

A.1 CAInterface class

This section describes the `CAInterface` class, which is the base class for all CADI interface classes.

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.
- Is 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 opaque 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.

These 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

This section describes the `CAInterface` class declaration.

```
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()

This section describes `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()

This section describes `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;
```

`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 CADIBroker class

This section describes the `CADIBroker` class, which enables connecting to existing simulations and creating new simulations.

A.2.1 CADIBroker class definition

This section describes the `CADIBroker` class definition.

```
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;
};
```

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. Delete the simulation 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).

A.2.2 Creating the CADIBroker

This is the first step in creating a new simulation or connecting to an existing one.

This example shows the prototypes for the functions that create the `CADIBroker`:

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();
}
```

A prototype declaration enables a global function to instantiate a broker from a dynamically loaded object:

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()

This method 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;
```

`startFactoryIndex`
is the index of the 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.


Note

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()

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

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

```
virtual CADIReturn_t CADIBroker::GetSimulationInfos(
                     uint32_t startSimulationInfoIndex,
                     uint32_t desiredNumberOfSimulations,
                     CADISimulationInfo_t *simulationList,
```

```
                        uint32_t *actualNumberOfSimulations) = 0;
```

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.

Array `simulationInfoList` must be at least this size.

---

simulationList

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

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;
```

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. 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()

This method releases 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 CADISimulationFactory class

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

A.3.1 CADISimulationFactory class definition

This section describes the `CADISimulationFactory` class definition.

```
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()

This method releases 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()

This method returns the name for this factory.

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

A.3.4 CADISimulationFactory::GetDescription()

This method returns the description for this factory.

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

A.3.5 CADISimulationFactory::GetParameterInfos()

This method 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;
```

`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.


Note

Array `parameterInfoList` must be 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 that is based on the given parameter values.

Errors occurring during system initialization are signaled through the given error callback `CADIErrorCallback`.


Note

This call can take a long 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;
```

`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. Do not call them 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 CADIErrorCallback class

This section describes the `CADIErrorCallback` class, which is the base class for error callback handlers that are addressed during instantiation.

A.4.1 CADIErrorCallback class definition

This section describes the `CADIErrorCallback` class definition.

```
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.4.2 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;
```

severity
is the severity of the error.

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.

Related information
CADIFactorySeverityCode_t on page 165

A.5 CADISimulationCallback class

This section describes the `CADISimulationCallback`, which is the base class for simulation callbacks. The class enables registering as a listener for system-wide callbacks.

A.5.1 CADISimulationCallback class definition

This section describes the `CADISimulationCallback` class definition.

```
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.5.2 CADISimulationCallback::simMessage()

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

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

message
is the message text to send to the listeners.

A.5.3 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.5.4 CADISimulationCallback::simKilled()

This callback is a last-ditch recovery method.

Suppose 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 CADISimulation class

This section describes the `CADISimulation` class, which represents a single simulation.

A.6.1 CADISimulation class definition

This section describes the `CADISimulation` class definition.

```
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;
```

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;
```

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;
```

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;
```

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 information 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;
```

targetID

is the id of the target to return.

A.7 CADICallbackObj class

This section describes the `CADICallbackObj` class, which is the base class for the `CADI` callbacks in the component.

A.7.1 CADICallbackObj class declaration

This section describes the CADICallbackObj class declaration.

```
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()

This method opens an application and returns the ID of the stream. CADI 2.0 deprecates this method.

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

sFileName
: is the 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 CsADICallbackObj::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;
```

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()

This method prints 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;
```

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()

This method closes the stream opened by `appliOpen()`. CADI 2.0 deprecates this method. Do not use it in new models.

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()

This method outputs 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()

This method reports a mode change from the target to the debugger.

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

newMode

is one of the `CADI_EXECMODE_*` constants.

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.

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


Note

Targets must support both callbacks to maintain backwards compatibility.

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

For an example implementation of this method, see `$PVLIB_HOME/examples/MTI/TraceOnBreak/`.

Related information

CADI_EXECMODE_t on page 189
CADICallbackObj::reset() on page 110

A.7.8 CADICallbackObj::reset()

This method 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.

Targets must support both callbacks to maintain backwards compatibility.

Arm recommends using `modeChange(CADI_EXECMODE_ResetDone)` in client code because a future version of CADI will deprecate the `reset()` callback.

Related information

CADICallbackObj::modeChange() on page 110

A.7.9 CADICallbackObj::cycleTick()

Arm deprecates this method. Do not use it.

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

A.7.10 CADICallbackObj::killInterface()

Arm deprecates this method. Do not use it.

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

A.7.11 CADICallbackObj::bypass()

This method is 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()

This method is 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
```

A target must not call this function while the simulation is running.

Related information
CADIRefreshReason_t on page 172
CADI_EXECMODE_t on page 189

A.8 CADI class

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

A.8.1 Methods in the CADI class

This section describes the methods in the `CADI` class that provide the main interfaces for configuring and running the target.

A.8.1.1 About the methods in the CADI class

This section describes common aspects of the methods in the CADI class.

For more details of the structs, enums, and defines that the CADI interface uses, see also the `CADITypes.h` file.

If called, unsupported methods must return `CADI_STATUS_CmdNotSupported`.

A.8.1.2 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.

A.8.1.3 Breakpoint API

The breakpoint API enables defining various types of breakpoint in the target model.

The types of breakpoint:

- Instruction execution.
- The content of a memory location.
- The content of a register.
- Temporary breakpoints for *run to* debugger behavior.
- Breakpoints on triggered exceptions.

A.8.1.4 Execution API

This section describes what the execution API enables a debugger to do.

- 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.

A.8.1.5 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.

A.8.1.6 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.

A.8.1.7 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.

A.8.1.8 Parameters API

This section describes what 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 Component CADI class declaration

This section describes the component CADI class declaration.

```
// 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;
};
```

Typically, you can leave the class declaration as it is, except for:

- 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 the `CADIDisassembler` and `CADIProfiler` classes for controlling and monitoring application execution.

Related information

CADIDisassembler class on page 146
CADIDisassemblerCB class on page 144
CADIProfiling class on page 151
CADIProfilingCallbacks class on page 150

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.

Related information

CADITargetFeatures_t on page 168

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;
```

`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 actual 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()

This deprecated method returns the `CADIDisassembler` for a target.

Arm deprecates obtaining disassemblers from CADI by calling `CADIGetDisassembler()`, but retains the method for compatibility with CADI 1.1. New code must call `ObtainInterface()` for both disassembler and profiling support.

```
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;
```

`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;
```

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;
```

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()

This method returns 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;
```

targetInfo

is set to point to the `CADITargetInfo_t` struct.

A.8.11 CADI::CADIGetParameterInfo()

This method gets the parameter info class for a specific parameter.

```
virtual CADIReturn_t CADIGetParameterInfo(const char *parameterName,
                                CADIParameterInfo_t *param) = 0;
```

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 is filled with data by the callee.


Note

For an example of using this method with `CADI::CADIGetParameterValues()` to retrieve parameter values, see A.8.12 Class reference - CADI::CADIGetParameterValues() on page 119

A.8.12 CADI::CADIGetParameterValues()

This method returns the current parameter values.

```
virtual CADIReturn_t CADI::CADIGetParameterValues(uint32_t parameterCount,
                              uint32_t *actualNumOfParamsRead,
                              CADIParameterValue_t *paramValuesOut) = 0;
```

parameterCount

is the length of array `paramValuesOut`.

actualNumOfParamsRead

is the number of valid entries in `paramValuesOut`. We recommend that the caller initializes this parameter to 0.

If an error code is returned and `actualNumOfParamsRead` is greater than 0, the first `actualNumOfParamsRead` entries are valid and caused no error. The entry `paramValuesOut[actualNumOfParamsRead]` caused the error.

paramValuesOut

points to a `CADIParameterValue_t` buffer. The caller must initialize its `parameterID` and `dataType` members, and the parameter values are filled by the callee.

Example A-1: Initializing paramValuesOut using CADI::CADIGetParameterInfo()

```
eslapi::CADIReturn_t status;
uint32_t desiredNumOfParams = 1;
uint32_t actualNumOfParams  = 0;
eslapi::CADIParameterInfo_t  paramInfo [desiredNumOfParams];
eslapi::CADIParameterValue_t paramValue[desiredNumOfParams]; // Caller must prepare
 this

// Caller must specify name of parameter to be retrieved
status = cadi->CADIGetParameterInfo("<parameter name>",&paramInfo);
if (status != eslapi::CADI_STATUS_OK) {
    // error handling
}

// Caller must set parameterID and dataType
paramValue[0].parameterID = paramInfo[0].id;
paramValue[0].dataType    = paramInfo[0].dataType;

status = cadi->CADIGetParameterValues(desiredNumOfParams, &actualNumOfParams,
 paramValue);
if (status != eslapi::CADI_STATUS_OK) {
    // error handling
}
```

Example A-2: Initializing paramValuesOut using CADI::CADIGetParameters()

```
eslapi::CADIReturn_t status;
uint32_t startIndex         = 0;
uint32_t desiredNumOfParams = 20;
uint32_t actualNumOfParams  = 0;
eslapi::CADIParameterInfo_t  paramInfo [desiredNumOfParams];
eslapi::CADIParameterValue_t paramValue[desiredNumOfParams]; // Caller must prepare
 this

status = cadi->CADIGetParameters(startIndex, desiredNumOfParams, &actualNumOfParams,
 paramInfo);
if (status != eslapi::CADI_STATUS_OK) {
    // error handling
}

for (unsigned i = 0; i < desiredNumOfParams; i++) {
    // Caller must set parameterID and dataType
    paramValue[i].parameterID = paramInfo[i].id;
    paramValue[i].dataType    = paramInfo[i].dataType;
}

status = cadi->CADIGetParameterValues(desiredNumOfParams, &actualNumOfParams,
 paramValue);
    if (status != eslapi::CADI_STATUS_OK) {
    // error handling
}
```

A.8.13 CADI::CADIGetParameters()

This method gets a 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;
```


Note

For an example of using this method with `CADI::CADIGetParameterValues()` to retrieve parameter values, see A.8.12 Class reference - CADI::CADIGetParameterValues() on page 119.

A.8.14 CADI::CADISetParameters()

This method sets parameter values.

```
virtual CADIReturn_t CADI::CADISetParameters(uint32_t parameterCount,
                                             CADIParameterValue_t *parameters,
                                             CADIFactoryErrorMessage_t *error) = 0;
```

A.8.15 CADI::CADIRegGetGroups()

This call retrieves register groups from the target.

```
virtual CADIReturn_t CADI::CADIRegGetGroups(uint32_t groupIndex,
                                            uint32_t desiredNumOfRegGroups,
                                            uint32_t *actualNumOfRegGroups,
                                            CADIRegGroup_t *reg) = 0;
```

`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()

This method obtains detailed register information.

Arm recommends 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;
```

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;
```

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;
```

regCount
is the requested number of registers (and consequently the size of the reg array).

reg
is an array of `CADIReg_t` structs 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` struct. 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;
```

regCount

is the number of requested registers and so the size of the reg array.

reg

is an array of `CADIReg_t` structs, each holding 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` struct. The `CADIRegGetMap()` call returns the number of registers.

numRegsRead

on return, is the number of registers 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 state 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.

CADI register access, showing how to set up the CADI access functions and test reading a register value

```
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()

This method 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::CADIGetCommitedPCs()

The method returns the number of program counters in the current cycle.

This method can be used with multi-issue processors.

```
virtual CADIReturn_t CADIGetCommitedPCs(int startIndex,
                                        int desiredCount,
                                        int *actualCount,
                                        uint64_t *pcs) = 0;
```

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()

Arm recommends 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;
```

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()

Arm recommends that the debugger for the target call this method once for each memory space, provided by calling the `CADIMemGetSpaces()` function, before accessing memory in that space.

This method 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;
```

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` structs. 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;
```

startAddress

is the start 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 large enough to hold the requested number of addresses. The target data has the same endianness as is declared in the memory space that is used for the access.

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 a side effect might be, for example, a clear-on-read.

If this parameter is set to `false`, the target must always omit side effects. This is a common use case when 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 to be 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;
```

startAddress

is the start 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 buffer contains target data that will be interpreted according the endianness that is declared in the memory space that is used for the access.

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 a side effect might be, for example, triggering an interrupt.

If set to `false`, the target must decide when to ignore this parameter. In 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 to be valid up to this position.
- If the write spans a gap in the memory space, the target must stop writing at 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;
```

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;
```

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;
```

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;
```

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;
```

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;
```

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 enum identifiers.

Related information

CADI_EXECMODE_t on page 189

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;
```

`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.

Related information

CADI_EXECMODE_t on page 189

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;
```

`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.

```
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()`.

Related information

CADI_EXECMODE_t on page 189

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;
```

`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()

This call provides a simulation level reset.

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.

```
virtual CADIReturn_t CADI::CADIExecReset(uint32_t resetLevel) = 0;
```

`resetLevel` must be one of the numbers provided in the `resetLevels` array received by `CADIExecGetResetLevels()`.

On receiving this call, the target resets 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 method causes the execution of the target to stop.

The method returns immediately and the target might still be running when the method 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;
```

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 method 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;
```

`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()

This method raises an exception.

```
virtual CADIReturn_t CADI::CADIExecAssertException(uint32_t exception,
                                    CADIExceptionAction_t action) = 0;
```

exception
is the exception number.

action
is the exception action to be taken.

Related information
CADIExceptionAction_t on page 191

A.8.43 CADI::CADIExecGetPipeStages()

This method 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;
```

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 method 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;
```

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 method 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;
```

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 method is used to unload symbol information of a specific image that was loaded previously.

```
virtual CADIReturn_t CADI::CADIExecUnloadApplication(const char *filename) = 0;
```

filename
is the same as for `CADIExecLoadApplication()`.

A.8.47 CADI::CADIExecGetLoadedApplication()

This method 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;
```

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 N^{th} 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 N^{th} 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;
```

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;
```

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.

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;
```

`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` structs used to return the requested breakpoints. The caller must allocate the array.

Related information

CADIBptDescription_t on page 188

A.8.51 Special purpose registers with permanent breakpoints for vector catching with CADIBptGetList()

Fast Models enables vector catching, using permanent breakpoints on special purpose registers. `CADIBptGetList()` returns these breakpoints, if present, in addition to temporary ones.

Table A-1: Cortex®-A and R special purpose registers with permanent breakpoints

Cortex-A and Cortex-R	TrustZone® (Non-secure)	TrustZone (Monitor)	Virtualization
RESET	-	-	-
UNDEFINED	NS_UNDEFINED	-	HYP_UNDEFINED
-	-	-	HYP_HYP
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: Cortex-A and R special purpose registers with permanent breakpoints unique to AArch64 processors
`EL` + *`descriptor`*, for example S_EL1_CURRENT_SP0_SYNC.

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

Exception levels				Descriptor
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

Table A-3: Cortex-M special purpose registers with permanent breakpoints, for exceptions

If TrustZone is not present	If TrustZone is present
RESET	RESET
LOCKUP	Secure_LOCKUP
IRQ[3]	Secure_IRQ and Non_Secure_IRQ[4]
NMI	Secure_NMI and Non_Secure_NMI
HARDFAULT	Secure_HARDFAULT and Non_Secure_HARDFAULT
MEMMANAGE	Secure_MEMMANAGE and Non_Secure_MEMMANAGE
BUSFAULT	Secure_BUSFAULT and Non_Secure_BUSFAULT
USGFAULT	Secure_USGFAULT and Non_Secure_USGFAULT
SVCALL	Secure_SVCALL and Non_Secure_SVCALL
DEBUG_MONITOR	Secure_DEBUG_MONITOR and Non_Secure_DEBUG_MONITOR
PENDSV	Secure_PENDSV and Non_Secure_PENDSV
-	SECUREFAULT
SYSTICK	Secure_SYSTICK and Non_Secure_SYSTICK

Table A-4: Cortex-M special purpose registers with permanent breakpoints, exception returns

If TrustZone is not present	If TrustZone is present
RETURN_LOCKUP	Secure_RETURN_LOCKUP
RETURN_IRQ[5]	Secure_RETURN_IRQ and Non_Secure_RETURN_IRQ[6]
RETURN_NMI	Secure_RETURN_NMI and Non_Secure_RETURN_NMI
RETURN_HARDFAULT	Secure_RETURN_HARDFAULT and Non_Secure_RETURN_HARDFAULT
RETURN_MEMMANAGE	Secure_RETURN_MEMMANAGE and Non_Secure_RETURN_MEMMANAGE
RETURN_BUSFAULT	Secure_RETURN_BUSFAULT and Non_Secure_RETURN_BUSFAULT
RETURN_USGFAULT	Secure_RETURN_USGFAULT and Non_Secure_RETURN_USGFAULT
RETURN_SVCALL	Secure_RETURN_SVCALL and Non_Secure_RETURN_SVCALL

[3] Catches all IRQs. To catch a specific IRQ, use register IRQn, where n is the IRQ number. These numbered registers do not have variants for different security states or for catching exception returns.
[4] Catch all Secure_IRQs or Non_Secure_IRQs. To catch a specific IRQ, use register IRQn, where n is the IRQ number. These numbered registers do not have variants for different security states or for catching exception returns.
[5] Catches all IRQ returns.
[6] Catch all Secure_IRQ or Non_Secure_IRQ returns.

If TrustZone is not present	If TrustZone is present
RETURN_DEBUG_MONITOR	Secure_RETURN_DEBUG_MONITOR and Non_Secure_RETURN_DEBUG_MONITOR
RETURN_PENDSV	Secure_RETURN_PENDSV and Non_Secure_RETURN_PENDSV
-	RETURN_SECUREFAULT
RETURN_SYSTICK	Secure_RETURN_SYSTICK and Non_Secure_RETURN_SYSTICK

A.8.52 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;
```

`breakpointId`
is the ID of the breakpoint to be read.

`requestOut`
is the return buffer for a single breakpoint.

A.8.53 CADI::CADIBptSet()

This method sets a code breakpoint in the target.

```
virtual CADIReturn_t CADI::CADIBptSet(CADIBptRequest_t *request,
                                      CADIBptNumber_t *breakpoint) = 0;
```

`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.8.54 CADI::CADIBptClear()

This method removes a breakpoint from the target.

```
virtual CADIReturn_t CADI::CADIBptClear(CADIBptNumber_t breakpointId) = 0;
```

`breakpointId`
is the requested breakpoint.


Note

`CADIBptClear()` returns `CADI_STATUS_IllegalArgument` for permanent breakpoints.

A.8.55 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;
```

`breakpointId`
is the requested breakpoint.

`configuration`
is the requested configuration.

A.9 CADIDisassemblerCB class

This section describes the `CADIDisassemblerCB` class and its methods.

A.9.1 CADIDisassemblerCB class definition

The disassembly front end must implement this callback 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;
```

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;
```

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;
```

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 CADIDisassembler class

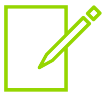
This section describes the CADIDisassembler class and its methods.

A.10.1 CADIDisassembler class definition

If a component supports disassembly, the Disassembly API can be used to display the disassembly during a simulation.


Note

Arm deprecates obtaining a disassembler from the CADI interface by calling `CADIGetDissambler()`. 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.

```
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. 32-bit versus 16-bit 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 enum:

```
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, 32-bit or 16-bit 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()

This method is used to obtain source-level information.

This method triggers the `CADIDisassemblerCB::ReceiveSourceReference()` callback.

```
virtual CADIDisassemblerStatus CADIDisassembler::GetSourceReferenceForAddress(
                                            CADIDisassemblerCB *callback,
                                            const CADIAddr_t &address) = 0;
```

`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;
```

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()

This method 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;
```

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` enum.

Related information

CADIDisassemblerStatus on page 192

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;
```

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 CADIProfilingCallbacks class

This section describes the `CADIProfilingCallbacks` class and its methods.

A.11.1 CADIProfilingCallbacks class definition

This section describes the `CADIProfilingCallbacks` class definition.

```
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()

This method profiles a resource access that has been registered by `CADIRegisterProfileResourceAccess()`.

```
virtual CADIReturn_t CADIProfilingCallback::profileResourceAccess(
                                const char *name,
                                CADIProfileResourceAccessType_t accessType) = 0;
```

`name`

is the name of the resource.

`accessType`

specifies the read/write access.

Related information

CADIProfileResourceAccessType_t on page 198

A.11.3 CADIProfilingCallbacks::profileRegisterHazard()

This method reports that a hazard of type `CADIProfileHazardDescription_t` has occurred.

```
virtual CADIReturn_t CADIProfilingCallback::profileRegisterHazard(
        CADIProfileHazardDescription_t desc) = 0;
```

`desc`

is of type `CADIProfileHazardDescription_t`.

Related information

CADIProfileHazardDescription_t on page 198

A.12 CADIProfiling class

The CADIProfiling class enables you to record and monitor profile information for debugging sessions. This section describes the class and its methods.

A.12.1 CADIProfiling class definition

This section describes the CADIProfiling class definition.

```
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 method informs the target of the memory regions that are to be profiled.

Call this function once before any number of calls to:

- `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;
```

`type`
is the type of profiling, execution addresses or data access, to which these regions apply. It is one of these values:

- `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. The caller allocates the required memory for this array.

The return value must be `CADI_STATUS_IllegalArgument` if any of these 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.

Related information

CADIProfileType_t on page 196
CADIProfileRegion_t on page 196

A.12.3 CADIProfiling::CADIProfileControl()

This method starts, stops, or resets profiling by passing a member of the `CADIProfileControl_t` enum.

```
virtual CADIReturn_t CADIProfiling::CADIProfileControl(
                      CADIProfileControl_t control) = 0;
```

control
defines profiling behavior.


Note

Starting profiling resets any saved information. Stopping profiling does not reset recorded information.

Related information
CADIProfileControl_t on page 196

A.12.4 CADIProfiling::CADIProfileTraceControl()

This method starts, stops, and resets recording the execution trace.

```
virtual CADIReturn_t CADIProfiling::CADIProfileTraceControl(
                                    CADITraceBufferControl_t bufferArg,
                                    CADITraceControl_t control,
                                    CADITraceOverlayControl_t overlay) = 0;
```

bufferArg
sets what to do when the buffer is full, that is, either wrap or stop.

control
defines the tracing behavior, and is one of these values:

- `CADI_TRACE_CNTL_StartContinuous`.
- `CADI_TRACE_CNTL_StartDiscontinuity`.
- `CADI_TRACE_CNTL_Stop`.

overlay
selects overlay mode, and is one of these values:

- 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.

Related information
CADITraceBufferControl_t on page 200
CADITraceControl_t on page 199
CADITraceOverlayControl_t on page 200

A.12.5 CADIProfiling::CADIProfileGetExecution()

This method 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;
```

`type`
indicates whether percentage statistics or an absolute count is being returned.

`regIndex`
is the index into the internal buffer that the target holds.

`regionSlots`
is the number of spaces that were 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 caller allocates and deallocates the array, which the target fills.

Related information

CADIProfileResultType_t on page 195
CADIProfileResults_t on page 195

A.12.6 CADIProfiling::CADIProfileGetMemory()

This method 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;
```

type
tells the caller whether percentage statistics or an absolute count is being returned.

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.

Related information

CADIProfileResultType_t on page 195
CADIProfileResults_t on page 195

A.12.7 CADIProfiling::CADIProfileGetTrace()

This method 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;
```

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.

Related information

CADITraceBlock_t on page 200

A.12.8 CADIProfiling::CADIProfileGetRegAccesses()

This method 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;
```

`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.

`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.

Related information

CADIProfileResults_t on page 195

A.12.9 CADIProfiling::CADIProfileSetRegAccesses()

This method 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;
```

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.


Note

`reg` must point to an array of objects of type `CADIResourceProfileResults_t` with size `numberOfRegs`.

actualNumberOfRegs
contains the number of actually updated registers.

Related information

CADIProfileResults_t on page 195

A.12.10 CADIProfiling::CADIProfileGetMemAccesses()

This method 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;
```

startAddress
is the start address for the selected memory units.

numberOfUnits
is the number of selected memory units.

mem
contains the results on return.


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.

Related information

CADIAddrComplete_t on page 182
CADIMemProfileResults_t on page 197

A.12.11 CADIProfiling::CADIProfileSetMemAccesses()

This method 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;
```

`startAddress`
is the starting address for the memory units.

`NumberOfUnits`
is the number of memory units.

`mem`
contains the values to use for the update of the profiling resources.

`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.

Related information

CADIAddrComplete_t on page 182
CADIMemProfileResults_t on page 197

A.12.12 CADIProfiling::CADIProfileGetAddrExecutionFrequency()

This method 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;
```

`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()

This method 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;
```

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()

This method returns the 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;
```

numberOfInstructions
is the required number of array entries to be prepared.

instructions
is an array that contains the values to use for preparing profiling.

actualNumberOfInstructions
is the number of array entries actually prepared.

Related information

CADIInstructionProfileResults_t on page 197

A.12.16 CADIProfiling::CADIProfileGetInstructionExecutionFrequency()

This method 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;
```

numberOfInstructions
is the required number of instructions to read to the profiling resources.

instructions
is an array to contain the results.

actualNumberOfInstructions
is the number of instructions actually read.

Related information

CADIInstructionProfileResults_t on page 197

A.12.17 CADIProfiling::CADIProfileSetInstructionExecutionFrequency()

This method 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;
```

numberOfInstructions
is the required number of array entries to write to the target.

instructions
contains the values to write to the target.

actualNumberOfInstructions
is the number of array entries actually written to the target.

Related information

CADIInstructionProfileResults_t on page 197

A.12.18 CADIProfiling::CADIRegisterProfileResourceAccess()

This method registers a resource access callback.

```
virtual CADIReturn_t CADIProfiling::CADIProfileRegisterResourceAccess(
                               const char *name,
                               CADIProfileResourceAccessType_t accessType) = 0;
```

name
is a resource.

accessType
is one of these values:

- `CADI_PROF_ACCESS_READ`.
- `CADI_PROF_ACCESS_WRITE`.
- `CADI_PROF_ACCESS_READ_OR_WRITE`.

Related information

CADIProfileResourceAccessType_t on page 198

A.12.19 CADIProfiling::CADIUnregisterProfileResourceAccess()

This method unregisters the resource access callback.

```
virtual CADIReturn_t CADIProfiling::CADIProfileUnregisterResourceAccess(
                                      const char *name) = 0;
```

A.12.20 CADIProfiling::CADIProfileRegisterCallBack()

This method registers a profiling callback to the target.

```
virtual CADIReturn_t CADIProfiling::CADIProfileRegisterCallBack(
                       CADIProfilingCallbacks *callBackObject) = 0;
```

callBackObject
is the callback.

Related information

CADIProfilingCallbacks class on page 150

A.12.21 CADIProfiling::CADIProfileUnregisterCallBack()

This method unregisters a profiling callback from the target.

```
virtual CADIReturn_t CADIProfiling::CADIProfileUnregisterCallBack(
                       CADIProfilingCallbacks *callbackObject) = 0;
```

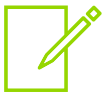
callBackObject
is the callback.

Related information

CADIProfilingCallbacks class on page 150

Appendix B Data structure reference

This appendix describes the data structures that CADI uses.


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

Most methods return this result. 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.

```
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,         // Argument 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 was not fulfilled due to a security violation
   CADI_STATUS_PermissionDenied,        // Request was not fulfilled since 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.

```
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 that
                                                        // caused this error is shown
                                                        // 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
                                                        // that prevented the CADI
                                                        // interface from being created
    CADIFACT_ERROR_GENERAL_WARNING,                     // Always warning = true, for
                                                        // everything else that still
                                                        // allowed the CADI interface
                                                        // to be created
    CADIFACT_ERROR_MAX = 0xFFFFFFFF
};
```

B.1.3 CADIFactorySeverityCode_t

The severity code is based on the error codes in `CADIFactoryErrorCode_t` and enables easy detection of errors and warnings.

```
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 struct contains details about a simulation.

```
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];
};
```

id

is for identification.

name

is the simulation name.

description

is the simulation description.

B.1.5 CADIParameterInfo_t

The `CADIParameterInfo_t` and `CADIParameterValue_t` structs 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];
};
```

id

is 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 runtime.

minValue
is the minimum admissible value.

maxValue
is the maximum admissible value.

defaultValue
if the type is `bool` or `int`, the default value.

defaultString
if the type is `CADI_PARAM_STRING`, the default string.

B.1.6 CADIParameterValue_t

The `CADIParameterInfo_t` and `CADIParameterValue_t` structs configure component parameters.

```
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];
};
```

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
if the type is `bool` or `int`, the integer value (0 = `false`, 1 = `true`).

maxValue
is the maximum admissible value.

stringValue

if the type is `string`, the string value.

B.1.7 CADITargetFeatures_t

The `CADIXfaceGetFeatures()` call uses the `CADITargetFeatures_t` struct.

```
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));
      }
```

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 that are 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 no program counter is 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 parameter is set to 0. Otherwise, this value can be a disjunction of these 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. These 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 memory space that the program counter points to.

SP_REGNUM=n

The ID of the register that is used as a stack pointer for the target architecture (or of a register with similar semantics).

LR_REGNUM=n

The ID of the register that is used as a link register for the target architecture (or of a register with similar semantics).

STATUS_REGNUM=n

The ID of the register that is 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.

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 execution, then expose `STACK_MEMSPACE_REGNUM`.

LOCALVAR_MEMSPACE_ID=n

The ID of the memory space that is used for storing local variables.

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 execution, then expose `LOCALVAR_MEMSPACE_REGNUM`.

`GLOBALVAR_MEMSPACE_ID=n`

The ID of the memory space that stores global vars.

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 execution, then expose `GLOBALVAR_MEMSPACE_REGNUM`.

`threadID=s`

If present, this parameter specifies the name of an implementation-specific mechanism for matching thread-aware breakpoint IDs. One possible value is CONTEXTIDR.

`HALT_CORE=n`

The ID of the register that halts or unclocks 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 or 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 organization specifies the string part: for example, `arm.com/my/mem/numbers`.

canCompleteMultipleInstructionsPerCycle

is `true` if the target can complete multiple instructions in a single simulation cycle.

Related information

Thread-aware breakpoints using CONTEXTIDR on page 187

B.1.8 CADICallbackType_t

The values in this type identify the different callback functions.

```
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 value is for input. Data travels from host to target
    CADI_CB_AppliOutput    = 2,  // This value is for output. Data travels from target to 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
};
```

These identifiers are, for example, used in the enable vector that is forwarded to the `CADIXfaceAddCallback()` call.

B.1.9 CADIRefreshReason_t

The target uses `CADI_REFRESH_REASON_t` constants to indicate why it has requested a refresh.

```
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, 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.

```
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;
};
```

`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 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 struct defines information about a register.

```
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;
};
```

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 parameter 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()`.

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, if the register has no DWARF register index `CADI_REGINFO_NO_DWARF_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 that is specified in `CADITargetFeatures_t::canonicalRegisterDescription`. If the scheme is the empty string, then no meaning can be ascribed to this field.

Related information

CADI::CADIRegGetCompound() on page 122

B.2.3 CADIRegDisplay_t

This section describes the register display values.

This enum 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.

```
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,      // You can use the program counter for disassembly display
    CADI_REGTYPE_BIN,     // Binary format
    CADI_REGTYPE_OCT      // Octal format
};
```

B.2.4 CADIRegSymbols_t

This struct is an array of symbolic values.

```
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

This enum determines the register access attribute value.

```
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 32 bits, not used
};
```

B.2.6 CADIRegType_t

This enum determines the register type.

```
enum CADIRegType_t
{
    CADI_REGTYPE_Simple,    // Register which has no subregisters.
    CADI_REGTYPE_Compound   // Register which has subregisters.
};
```

B.2.7 CADIRegDetails_t

This struct defines the register details.

```
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

This struct defines the register group.

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;
    };
```

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

This struct contains 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;
};
```

`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 or a Physical space.

isCache
specifies that this memory space is a cache.

endianness
is the endianness, 0 = variable endianness as defined by the architecture, 1 = always little-endian, 2 = always big-endian.

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 that is 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 struct 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.

```
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;
};
```

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 that are required for an access to this block.

readWrite
specifies the read/write type of this block.

supportedMultiplesOfMAU
indicates the multiples of 1 byte. If for example the MAU size is 8 bits and the supported access is 32 bits, the corresponding value is 4 (from 32 bits or 8 bits).

endianness
is the endianness, 0 = same as owning memory space, 1 = little-endian, 2 = big-endian.

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.

```
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;
};
```

space
is the numeric designation of the memory space (`uint32_t`).

addr
is the actual memory address (`uint64_t`).

B.2.12 CADIMemReadWrite_t

This enum signifies the read and write status for a block of memory.

```
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.

```
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;
};
```

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

This struct contains cache info data.

```
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;
};
```

cacheLineSize

is the size of a cache line 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.

```
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;
};
```

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 whether or not 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 these 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 or memory has been modified. This trigger might be the result of an explicit register or memory access or (for example in case of registers or 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.

`triggerType` only has meaning for `CADI_BPT_REGISTER` and `CADI_BPT_MEMORY` breakpoints:

- The debugger must set `triggerType` 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. All types of breakpoint must obey this field, including `CADI_BPT_INST_STEP`.

B.3.2 CADIBptCondition_t and CADIBptConditionOperator_t

This section describes the CADIBptCondition_t and CADIBptConditionOperator_t structs.

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.

If the `useFormalCondition` is set, `CADI_BPT_PROGRAM`, `CADI_BPT_PROGRAM_RANGE`, `CADI_BPT_INST_STEP`, `CADI_BPT_EXCEPTION` must obey the `ignoreCount`. 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;
};
```

conditionOperator

specifies the types of condition that determine whether a breakpoint matches. It specifies how the fields `comparisonValue` and `threadID` are interpreted. If this field is set to a value that the target does not support, targets must return `CADI_STATUS_ArgNotSupported` from `CADIBptSet()`.

comparisonValue

if the bottom 30-bits of `conditionOperator` have a value other than `CADI_BPT_COND_UNCONDITIONAL (0)`, then they specify how to compare `comparisonValue` with the value associated with the breakpoint hit. See the list of the enumerated condition values for `CADIBptConditionOperator_t`.

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 to match thread IDs. Targets must use the `extendedTargetFeatures` register to identify the mechanism being used. (See the `CADITargetFeatures_t` struct under the `nExtendedTargetFeaturesRegNum` entry.) One possible mechanism for Arm targets is to match against `CONTEXTIDR`. If `threadID` is nonzero and bit 31 of `conditionOperator` is not set, targets must return `CADI_STATUS_ArgNotSupported` from `CADIBptSet()`.

ignoreCount

is the number of breaks to ignore.

bitwidth

is the width of comparison value.

The conditional breakpoint operations are enumerated in `CADIBptConditionOperator_t`.

```
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,

    CADI_BPT_COND_THREADID = 0x80000000,   // Thread-aware breakpoint

        // these are no breakpoint conditions:
    CADI_BPT_COND_ENUM_MAX = 0xFFFFFFFF
};
```

Related information

CADITargetFeatures_t on page 168
Thread-aware breakpoints using CONTEXTIDR on page 187

B.3.3 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 as an `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. If bit 31 of `conditionOperator` is 0, the `threadID` field is ignored.

Related information

CADITargetFeatures_t on page 168

B.3.4 CADIBptDescription_t

This section describes the `CADIBptDescription_t` struct.

```
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;
};
```

`bptNumber`

is the breakpoint number (`uint32_t`).

`bptInfo`

is the breakpoint information such as address or condition.

B.3.5 CADIBptConfigure_t

This section describes the definition of `CADIBptConfigure_t`.

```
enum CADIBptConfigure_t
  {
    CADI_BPT_Disable,
    CADI_BPT_Enable
  };
```

B.3.6 CADIExecMode_t

This struct returns the execution mode.

```
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];
}
```

number

indicates the execution mode and must be one of the `CADI_EXECMODE_t` values.

name

is the mode name.

Related information

CADI_EXECMODE_t on page 189

B.3.7 CADI_EXECMODE_t

This section describes the values in the `CADI_EXECMODE_t` enum.

```
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    };
```

`modeChange()` uses the enum values:

modeChange(CADI_EXECMODE_Stop)

The simulation was in state “running” and has now stopped. This callback is always the last one in a sequence of callbacks when the simulation stopped. If the stop was because one or more breakpoints have been hit, then this callback follows one or more `modeChange(CADI_EXECMODE_Bpt, num)` callbacks where `num` is the breakpoints 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 that 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 precede 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 callback is the same as `modeChange(CADI_EXECMODE_Stop)`, but the model is in a state "stopped and error" after this callback. Consequently, all execution control functions are disabled. `CADIExecReset()` must be called first to enable them again. This callback does not precede 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 that registers and memory have changed.

modeChange(CADI_EXECMODE_ResetDone)

The `CADIExecReset()` request that was recently requested by a debugger is now complete. This callback is always the last one in a sequence of callbacks that are caused by a `CADIExecReset()`. If the model was running when `CADIExecReset()` was issued, a `modeChange(CADI_EXECMODE_Stop)` might happen before this callback.

`CADIExecReset()` is an asynchronous call. Each debugger that is 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 that registers and memory have changed.

B.3.8 CADIResetLevel_t

This section describes the definition of `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.3.9 CADIException_t

This section describes the definition of `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.3.10 CADIExceptionAction_t

This section describes the definition of `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.

```
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;
};
```

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` enum.

B.4.2 CADIPipeStageContentInfo_t

This section describes the definition of `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

This section describes the `CADIDisassemblerStatus` enum.

```
enum CADIDisassemblerStatus
{
   CADI_DISASSEMBLER_STATUS_OK,              // Disassembling completed successfully.
   CADI_DISASSEMBLER_STATUS_NO_INSTRUCTION,  // Current address points to illegal instructions
                                             // or data.
   CADI_DISASSEMBLER_STATUS_ILLEGAL_ADDRESS, // Address out of range (memory read failed).
   CADI_DISASSEMBLER_STATUS_ERROR            // Other error.
};
```

B.5.2 CADIDisassemblerType

This section describes the `CADIDisassemblerType` enum.

```
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

This section describes the `CADIDisassemblerInstructionType` enum.

```
enum CADIDisassemblerInstructionType
{
   CADI_DISASSEMBLER_INSTRUCTION_TYPE_NOCALL,  // The instruction is not a call, so e.g. 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.

```
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.

```
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.

```
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;
};
```

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 per line basis.

B.6.4 CADIStreamId

This set of `streamIds` is reserved for special cases.

These cases are the special ones:

```
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. Do not do so.

```
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

This enum enables the target to specify whether the results represent a percentage of the whole or a total count.

```
enum CADIProfileResultType_t
    {
        CADI_PROF_RESULT_Percentage,
        CADI_PROF_RESULT_Count
    };
```

B.7.2 CADIProfileResults_t

Objects of this type contain the results of a profiling session.

```
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

This section defines `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 another region does not include.

Two overlays for the same memory addresses do not constitute a shared memory space.

```
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

This enum determines the type of profiling to which the region definition applies.

```
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

This enum describes the action the call is trying to apply to the target profiling mechanism.

```
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 enum defines the accesses that are permitted for the resource.

```
enum CADIProfileResourceAccessType_t
{
        CADI_PROF_ACCESS_READ,
        CADI_PROF_ACCESS_WRITE,
        CADI_PROF_ACCESS_READ_OR_WRITE
};
```

B.7.10 CADIProfileHazardTypes_t

This enum defines hazard information for the resource.

```
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.

```
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];
};
```

type

is the number of accesses to affected resource.

numberOfAccesses

is the FID of the originator resource or instruction.

affectedInstructionFID

is the name of the affected resource or instruction.

resource

is the resource.

message

is the hazard message.

B.7.12 CADITraceControl_t

This enum describes the type of control being exerted on the trace mechanism.

```
enum CADITraceControl_t
    {
        CADI_TRACE_CNTL_StartContinuous,
        CADI_TRACE_CNTL_StartDiscontinuity,
        CADI_TRACE_CNTL_Stop
    };
```

B.7.13 CADITraceBufferControl_t

This enum describes the type of control being exerted on the trace mechanism.

```
enum CADITraceBufferControl_t
    {
        CADI_TRACE_BUFF_Wrap,
        CADI_TRACE_BUFF_StopOnFull
    };
```

B.7.14 CADITraceOverlayControl_t

This enum describes the type of control being exerted on the trace mechanism.

```
enum CADITraceOverlayControl_t
    {
        CADI_TRACE_OVERLAY_Manager,
        CADI_TRACE_OVERLAY_Memory
    };
```

B.7.15 CADITraceBlockType_t

This enum describes the type of data in a `CADITraceBlock_t`.

```
enum CADITraceBlockType_t
    {
        CADI_TRACE_BLK_Address,
        CADI_TRACE_BLK_Overlay
    };
```

B.7.16 CADITraceBlock_t

This struct describes a single piece of trace data that either contains an overlay ID or an address.

```
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;
};
```