Iris Developer Guide

Copyright © 2018–2020 Arm Limited or its affiliates. All rights reserved.

Release Information

<table>
<thead>
<tr>
<th>Issue</th>
<th>Date</th>
<th>Confidentiality</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100-00</td>
<td>23 November 2018</td>
<td>Non-Confidential</td>
<td>New document.</td>
</tr>
<tr>
<td>0100-01</td>
<td>26 February 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.6.</td>
</tr>
<tr>
<td>0100-02</td>
<td>17 May 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.7.</td>
</tr>
<tr>
<td>0100-03</td>
<td>05 September 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.8.</td>
</tr>
<tr>
<td>0100-04</td>
<td>28 November 2019</td>
<td>Non-Confidential</td>
<td>Update for v11.9.</td>
</tr>
<tr>
<td>0100-05</td>
<td>12 March 2020</td>
<td>Non-Confidential</td>
<td>Update for v11.10.</td>
</tr>
</tbody>
</table>

Non-Confidential 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 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.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms. 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 subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm’s trademark usage guidelines at http://www.arm.com/company/policies/trademarks.

Copyright © 2018–2020 Arm Limited (or its affiliates). All rights reserved.
# Contents

**Iris Developer Guide**

## Preface

**About this book** ...................................................... ......................................................  7

## Chapter 1 Iris overview

1.1 Overview .................................................................................................................. 1-10
1.2 References .............................................................................................................. 1-11
1.3 Terms and abbreviations ...................................................................................... 1-12
1.4 Interfaces and communication ............................................................................. 1-13
1.5 Iris specification changelog ................................................................................. 1-16
1.6 IrisSupportLib changelog .................................................................................... 1-17

## Chapter 2 Iris examples

2.1 C++ client examples ............................................................................................ 2-19
2.2 DummyModel example ....................................................................................... 2-20
2.3 Plug-in examples .................................................................................................. 2-21
2.4 Python examples .................................................................................................. 2-22

## Chapter 3 Generic function call interface

3.1 JSON data types .................................................................................................... 3-24
3.2 JSON-RPC 2.0 function call format ..................................................................... 3-26
3.3 Synchronous and asynchronous behavior ......................................................... 3-28
3.4 Sending a request, a notification, and a response ............................................. 3-29
3.5 U64JSON ............................................................................................................. 3-31
3.6 Function call optimizations ............................................................................... 3-39
Chapter 4  
Object model
4.1 Object model overview ............................................................... 4-56
4.2 Instances .................................................................................. 4-57

Chapter 5  
Iris APIs
5.1 Iris API documentation ............................................................ 5-60
5.2 Naming conventions ................................................................ 5-61
5.3 instId argument ........................................................................ 5-62
5.4 Compatibility rules for function callers and callees .................. 5-63
5.5 Iris-text-format ......................................................................... 5-64
5.6 Resources API ......................................................................... 5-67
5.7 Memory API ............................................................................. 5-88
5.8 Disassembly API ...................................................................... 5-104
5.9 Tables API ................................................................................ 5-109
5.10 Image loading and saving API .................................................. 5-115
5.11 Simulation time execution control API ...................................... 5-123
5.12 Debuggable state API .............................................................. 5-126
5.13 Stepping API .......................................................................... 5-130
5.14 Per-instance execution control API ......................................... 5-134
5.15 Breakpoints API ...................................................................... 5-136
5.16 Notification and discovery of state changes API ...................... 5-144
5.17 Events and trace API ............................................................... 5-145
5.18 Semihosting API ..................................................................... 5-161
5.19 Simulation accuracy (sync levels) API ..................................... 5-167
5.20 Instance registry, instance discovery, and interface discovery API 5-172
5.21 Simulation instantiation and discovery API ............................ 5-189
5.22 Plug-in loading and instantiation API ....................................... 5-197
5.23 TCP server management API .................................................. 5-199
5.24 Checkpointing API .................................................................. 5-201

Chapter 6  
Response error codes
6.1 Function-specific error codes .................................................... 6-203
6.2 Function-independent error codes .......................................... 6-208
This preface introduces the *Iris Developer Guide*. It contains the following:

- *About this book on page 7.*
About this book

This book describes the Iris interface for debug and trace on Fast Models and other targets. Iris defines a generic function call mechanism, an object model, and a set of concrete functions for debug and trace.

Using this book

This book is organized into the following chapters:

Chapter 1 Iris overview
This chapter describes the purpose and implementation of Iris.

Chapter 2 Iris examples
This chapter describes the examples of Iris clients, plug-ins, and Python scripts. Each client and plug-in example contains source code and a makefile for GCC or project file for Microsoft Visual Studio.

Chapter 3 Generic function call interface
Iris interfaces are named functions that receive named arguments. The functions mostly receive and return structured data as objects that contain named values. In case of an error, they return an error code. This chapter describes how to access Iris interfaces from C++ or using IPC/TCP.

Chapter 4 Object model
Iris provides an object model in which all entities are represented by instances. Instances can discover other instances and can call functions on each other. For example, a debugger can read a register in a CPU model, and the CPU model can send trace data to the debugger. Both debugger and model are instances.

Chapter 5 Iris APIs
This chapter describes the Iris APIs. It provides conceptual information for each API, followed by detailed reference information for each function, object, and event source in the API.

Chapter 6 Response error codes
This chapter describes the error codes that Iris defines. These are in addition to those defined in the JSON-RPC 2.0 specification.

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.

Typographic conventions

*italic*
Introduces special terminology, denotes cross-references, and citations.

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

*monospace*
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

*monospace* *
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

*monospace* *italic*
Denotes arguments to monospace text where the argument is to be replaced by a specific value.
**monospace bold**
Denotes language keywords when used outside example code.

*<and>*
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**
Used in body text for a few terms that have specific technical meanings, that are defined in the Arm® Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

**Feedback**

**Feedback on this product**
If you have any comments or suggestions about this product, contact your supplier and give:
- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

**Feedback on content**
If you have comments on content then send an e-mail to errata@arm.com. Give:
- The title *Iris Developer Guide*.
- The number 101196_0100_05_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

**Note**
Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

**Other information**
- Arm® Developer.
- Arm® Information Center.
- Arm® Technical Support Knowledge Articles.
- Technical Support.
- Arm® Glossary.
Chapter 1
Iris overview

This chapter describes the purpose and implementation of Iris.

Iris consists of:

- A generic function call interface that uses JSON-RPC 2.0 format and semantics.
- A simple object model in which all entities, for example components, debuggers, clients, and plug-ins are represented by instances. Instances can discover and communicate with all other instances.
- A defined set of functions for debug and trace.

It has several benefits over previous debug and trace solutions:

- Network native. Both simulation control and trace are available over the network.
- Plug-ins and trace can be loaded at any point during the simulation.
- Guaranteed synchronisation between trace and simulation control, when required.
- Improvements to debug APIs. Iris provides:
  — Asynchronous trace.
  — Address translation.
  — Table API.
- Extensibility. New functionality can be added without breaking compatibility.
- Improvements to debug functionality offered by components.

It contains the following sections:

- 1.1 Overview on page 1-10.
- 1.2 References on page 1-11.
- 1.3 Terms and abbreviations on page 1-12.
- 1.4 Interfaces and communication on page 1-13.
- 1.5 Iris specification changelog on page 1-16.
- 1.6 IrisSupportLib changelog on page 1-17.
1.1 Overview

The Iris interface consists of the following:

- A generic function call interface. Functions are called by name. Arguments and return values are passed by name and by value.
- An object model. Iris systems consist of a set of instances. Instances are entities that can send and receive Iris function calls, for instance modeled components, hardware targets, debuggers, plug-ins, and framework components. All instances can:
  — Discover and communicate with all other instances.
  — Call functions on and receive function calls from all other instances.

Each instance is identified by an instance id, \texttt{instId}.

- A defined set of functions that are supported by instances. Most are optional.

It has the following design principles:

- Function calls are generally split into a request and an asynchronous response.
- Function calls and responses are represented using JSON data types, for example integers, strings, arrays, objects, and booleans.
- All types of events, for instance trace, debug, and semihosting events, are exposed through the same event mechanism.
- When calling a function, the caller can generally choose between sending:
  — A request. The callee sends a response back to the caller when it has finished processing the function. This is called a synchronous or blocking function call, as the caller is blocked while waiting for the response.
  — A notification. The callee does not send a response, so the caller does not know when the callee has finished. This is called an asynchronous or non-blocking function call. It is faster than a request and is preferred.
- Instances can connect to a simulation either using Inter-Process Communication (IPC) or within the same process. In either case, the interface is the same.
- JSON RPC 2.0 semantics are used for all function calls, whether using IPC or in-process.
- It uses U64JSON, a proprietary binary equivalent of JSON that is based on a sequence of \texttt{uint64_t} values, in-process to remove the JSON parsing overhead.
## 1.2 References

This document refers to the following websites:

<table>
<thead>
<tr>
<th>URL</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://json.org/index.html">http://json.org/index.html</a></td>
<td>Introducing JSON</td>
</tr>
<tr>
<td><a href="http://www.jsonrpc.org/specification">http://www.jsonrpc.org/specification</a></td>
<td>JSON-RPC 2.0 Specification</td>
</tr>
<tr>
<td><a href="http://www.simple-is-better.org/json-rpc/transport_http.html">http://www.simple-is-better.org/json-rpc/transport_http.html</a></td>
<td>JSON-RPC 2.0 Transport: HTTP</td>
</tr>
</tbody>
</table>
1.3 Terms and abbreviations

This table defines some terms and abbreviations that are commonly used in the Iris documentation, or have a meaning that is Iris-specific:

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iff</td>
<td>Short for <em>if and only if</em>.</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication, either on the same host, for example TCP or pipes, or on different hosts, for example TCP.</td>
</tr>
<tr>
<td>DSO</td>
<td>Dynamic Shared Object (* .so) or DLL.</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation. A compact textual representation of data.</td>
</tr>
<tr>
<td>JSON RPC</td>
<td>Remote Procedure Call protocol using JSON.</td>
</tr>
<tr>
<td>U64JSON</td>
<td>Binary JSON variant that is based on arrays of 64-bit values. It is defined in 3.5 U64JSON on page 3-31.</td>
</tr>
<tr>
<td>Component</td>
<td>A piece of software with well-defined abstract interfaces that represents a piece of hardware or other functionality.</td>
</tr>
<tr>
<td>Target</td>
<td>Sometimes used as a synonym for <em>component</em> or <em>instance</em> but generally avoided in this document.</td>
</tr>
<tr>
<td>Instance</td>
<td>An entity that provides functionality to other instances, or uses functionality that is provided by other instances, or both. For example, components, debuggers, clients, and plug-ins are all instances.</td>
</tr>
<tr>
<td>Event</td>
<td>Any event that is produced by an instance. This might be a trace event, for example INST for each executed instruction, a simulation event, for example IRIS_BREAKPOINT_HIT, or any other kind of event. Clients can observe events upon request.</td>
</tr>
<tr>
<td>Iris</td>
<td>An interface for debug and trace. Not an abbreviation or acronym.</td>
</tr>
</tbody>
</table>
1.4 Interfaces and communication

The following diagram provides an overview of the Iris interfaces:

![Diagram of Iris architecture]

--- Iris function calls encoded in U64JSON (bi-directional)
----- Iris function calls encoded in JSON, U64JSON, or other supported formats across TCP (bi-directional)

**Figure 1-1 Iris architecture**

An Iris system consists of the following:

**Simulator executable**

The simulator executable can be implemented using C or C++, SystemC, Gem5, or Fast Models. It can be a standalone executable or a DSO.

**Iris instances**

The term *Iris instance*, or just *instance*, refers to an entity that can send and receive Iris function calls. This includes all components, plug-ins, clients, for example debuggers, and framework entities, for example the global instance. All Iris instances can send and receive Iris function calls to and from all other instances. In *Figure 1-1 Iris architecture on page 1-13*, Iris instances are shown by boxes with a bold outline.

**IrisSupportLib**

A static library named IrisSupport.lib or libIrisSupport.a that provides the following core Iris functionality:

- The global instance and the global instance registry.
- Routes all Iris messages.
- Plug-in loading.
- The IrisTcpServer.

It is linked and managed by the model. IrisSupportLib is documented in *IrisSupportLib Reference Manual*.

**framework.GlobalInstance**

This is the central routing instance between all Iris instances. The global instance registry records all Iris instances in the system. All Iris instances must register and unregister themselves in the instance registry, and they can use it to query a list of all other instances.
**IrisTcpServer**

TCP server that is provided by IrisSupportLib and runs as part of the simulator executable. It listens for connections from TCP clients, typically an IrisTcpClient. It transparently forwards function calls and responses. It does not explicitly support Iris functions, so extending the Iris interface, for example by adding functions or arguments, or adding new data structures, does not require any changes to the IrisTcpServer.

The IrisInterface communication trivially maps onto a TCP socket because it is inherently split between request and response, and because function calls and responses are data only. This data is transmitted almost unchanged over TCP between clients and servers.

**IrisC API**

A C interface that is used on DSO boundaries. It is equivalent to the C++ IrisInterface.

**U64JSON**

A proprietary binary variant of JSON, which is JSON-compatible. It is based on uint64_t arrays and is optimized for speed, not size. It removes the runtime overhead of JSON parsing and data conversion. It is used in-process and is one of many options for out-of-process, or IPC communication.

**IrisInterface**

An in-process, generic mechanism that transports Iris function calls, including callbacks and responses. In-process function calls and responses are made according to the JSON RPC 2.0 specification and are encoded in U64JSON. Instance implementations usually use a helper class, IrisInstance, which hides the internals of IrisC, IrisInterface, and U64JSON.

**IrisInstance**

Implements all necessary boilerplate code to provide debuggers and components with easy access to Iris functions. IrisSupportLib provides implementations for C++ and Python. For example, it provides:

- Encoding and decoding of function calls.
- Blocking function call semantics.
- Data in native data types for the language being used.
- Generic error messages.

**IrisTcpClient**

TCP client that is provided by IrisSupportLib. As with the IrisTcpServer, extending the Iris interface does not require any changes to the IrisTcpClient. Using the IrisTcpClient to connect to the IrisTcpServer is not mandatory, but is convenient. A client application, for example a debugger, typically uses an IrisInstance connected to an IrisTcpClient to connect to an Iris server running in the model process.

**Iris function calls over TCP**

The protocol that is used over TCP and the format and semantics of all Iris functions are defined and public. In Figure 1-1 Iris architecture on page 1-13, they are shown by dashed lines.

**client.debugger**

A C++ client application that uses IrisSupportLib, built from source. It can call Iris functions directly from C++ and can update the IrisSupportLib source at any time. An update is not mandatory after the simulator executable has updated any part of the system. Clients and simulators can update at different times. There are no shared header files, but both sides must follow the Iris specification to be compatible.

**client.PythonScript**

The same as client.debugger but written in Python.
**component.Plugin0 and component.Plugin1**

These plug-ins use IrisInstance to communicate with the rest of the system. There is no difference between a plug-in and a client that is connected using IPC in how they call and are called by Iris functions, except for the plug-in loading mechanism and speed considerations.

--- **Note** ---

Plug-ins can communicate with each other in the same way as with the rest of the system, including with clients connected using IPC.

**component.Comp0-2**

Components written in C++, LISA, or SystemC, that model hardware or perform other simulation functionality. They use IrisInstance to avoid being exposed to the internals of the function call mechanism, and to use infrastructure that is common to a lot of components, for example meta information for registers and memory spaces. Internally in the IrisInstance, they send and receive U64JSON-encoded Iris function calls through the low-level IrisC API. They might buffer these function calls in an event queue. Later on, and typically from another thread, they send the response back to the IrisInterface of the global instance.

**Transports**

Iris function calls are transported using the following mechanisms:

- **In-process.** The transport is the IrisC interface on DSO boundaries and the C++ IrisInterface inside DSOs. The Iris interface is bi-directional so it can send function calls and responses in both directions. IrisInterface only supports U64JSON directly, but adapters exist to enable function calls from C++ and Python directly, or to use other formats, for example JSON. The in-process mechanism is used whenever possible, typically by:
  - Plug-ins that communicate with component instances, for example to receive trace and to inspect components.
  - The global instance to communicate with component instances. For example, components register themselves with the global instance at startup.
  - The IrisTcpServer to communicate with component instances and with the global instance.
- **Inter Process Communication** (IPC). The transport is a TCP socket. Usually the simulation contains a TCP server which listens for inbound connections. Iris calls and their responses are sent in both directions across the same TCP socket. The TCP connection is persistent.

Various formats can be used across the TCP connection, including U64JSON and JSON. IPC is used only if necessary, typically by IPC clients, for example debuggers, shells, and IPC plug-ins, to communicate with component instances, the global instance, plug-ins, or even other clients.

**Related information**

IrisSupportLib Reference Manual
1.5 Iris specification changelog

Lists all relevant changes to the Iris specification, especially all changes to Objects/*.json and Functions/*.json.

--- Note ---

This topic does not list user-visible changes to IrisSupportLib. For these changes see 1.6 IrisSupportLib changelog on page 1-17.

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-04-01</td>
<td>Changed the way register fields are exposed. Added ResourceInfo.parentRscId on page 5-80 which if present indicates that this is a child resource. It contains the resource id of the parent resource. Both name and cname in ResourceInfo no longer contain the hierarchical name of the child register but instead just the name of the child register itself (interface change), without any parent register names prepended. A dot in the name or cname members no longer indicates a parent/child relationship. This change is not fully backwards compatible with old clients, but the consequences of breaking the interface are so minor (only affects the naming of child registers) and the old behavior was so obscure that it can be assumed that no clients relied on it anyway.</td>
</tr>
<tr>
<td>2019-04-03</td>
<td>Made ResourceInfo.cname on page 5-80 and ResourceGroupInfo.cname on page 5-79 non-optional to simplify client behavior and clarify expectations. Existing clients are unaffected. Clients can now rely on cname always being present.</td>
</tr>
<tr>
<td>2019-04-04</td>
<td>Clarified that each resource is either a register or a parameter.</td>
</tr>
<tr>
<td>2019-04-16</td>
<td>Added ResourceTags.isFramePointer on page 5-85.</td>
</tr>
<tr>
<td>2019-04-23</td>
<td>Clarified that parameters are not hierarchical (there are no child parameters). Specified that all parameter resources must be in group Parameters.</td>
</tr>
<tr>
<td>2019-08-01</td>
<td>Added function instance_getCppInterfaceIrisInstance() on page 5-181 to allow enhancing IrisInstances with tightly coupled plugins.</td>
</tr>
<tr>
<td>2019-08-02</td>
<td>Marked the following arguments as optional for consistency as originally intended: 'AttributeInfo.descr', all EventCounterMode fields, 'InstantiationError.parameterName', 'TableColumnInfo.formatLong', 'TableColumnInfo.formatShort' and 'MemoryReadResult.error'.</td>
</tr>
<tr>
<td>2019-08-02</td>
<td>Consistency fixes: Removed old leftover field 'EventSourceInfo.stop'. Fixed 'ResourceInfo.subRscId' (was erroneously called subResId in the specification).</td>
</tr>
<tr>
<td>2019-10-25</td>
<td>Consistency fix: Renamed descr to description everywhere.</td>
</tr>
<tr>
<td>2019-10-28</td>
<td>Removed 'IrisInterface::irisObtainInterface()' since it was not used and is superseded by Iris getCppInterfaceXYZ() calls.</td>
</tr>
</tbody>
</table>
1.6 IrisSupportLib changelog

Lists changes to the IrisSupportLib implementation (except for minor changes).

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2019-03-21</td>
<td>Bugfixes: Properly initialized all optional members in objects returned through an output reference in Iris C++ functions (for example, ResourceReadResult.error in resource_read()). Made TableColumnInfo.rwMode optional. Clients might now receive an empty string which means rw as specified. Made TableColumnInfo.indexFormatHint optional. Clients might now receive an empty string which means hex as specified.</td>
</tr>
<tr>
<td>2019-04-05</td>
<td>Cleaned-up building parameter resources: Replaced addResource().setParameterInfo() with addParameter() in IrisInstanceBuilder. Same for string parameters. Removed functions addResource(), addStringResource() and addNoValueResource(). Each resource is either a register or a parameter and must be added using addRegister() or addParameter(), respectively.</td>
</tr>
<tr>
<td>2019-04-11</td>
<td>Removed defaults for register. canonicalRnScheme and memory. canonicalMsnScheme. Instance implementations now need to set these explicitly.</td>
</tr>
<tr>
<td>2019-04-15</td>
<td>Added support for canonical register number scheme ElfDwarf. In particular added 'include/iris/detail/IrisElfDwarf.h', 'include/iris/IrisElfDwarfArm.h' and 'IrisInstanceBuilder::RegisterBuilder::setCanonicalRnElfDwarf()'.</td>
</tr>
<tr>
<td>2019-04-16</td>
<td>Child registers no longer implicitly inherit canonicalRn and addressOffset from their parents.</td>
</tr>
<tr>
<td>2019-04-23</td>
<td>Added options to iris_inspect.py: --cpus-only, --first-only, --canonicalRn-table (generate a table of all registers that have a canonicalRn assigned), --properties, --resource-groups and -1 (single line details).</td>
</tr>
<tr>
<td>2019-04-23</td>
<td>Added IrisElfDwarfArm.h header which defines canonicalRns for ARM cores.</td>
</tr>
<tr>
<td>2019-04-23</td>
<td>Added 'ResourceTags.isFramePointer'.</td>
</tr>
<tr>
<td>2019-04-29</td>
<td>Added support for out-of-process trace clients to Examples/Plugin/SimpleTrace.</td>
</tr>
<tr>
<td>2019-06-13</td>
<td>Added disassembly and target property support to Python/Examples/IrisDebugger.py.</td>
</tr>
<tr>
<td>2019-06-19</td>
<td>Added IrisInstanceSemihosting::unblock() to allow unblocking of blocked semihosting requests.</td>
</tr>
<tr>
<td>2019-06-20</td>
<td>Added handle_semihosting_io() to Python iris. debug to explicitly request handling semihosting output in Python. The default is to continue to print it to the console.</td>
</tr>
<tr>
<td>2019-06-20</td>
<td>Removed IrisInstanceSyncLevel without replacement as it was not applicable to the sole use-case.</td>
</tr>
<tr>
<td>2019-06-24</td>
<td>Added sync level and partial tables support to Python/Examples/IrisDebugger.py</td>
</tr>
<tr>
<td>2019-07-02</td>
<td>Fixed deadlock in IrisTcpClient.java.</td>
</tr>
<tr>
<td>2019-07-05</td>
<td>Added plugin support to Python/Examples/IrisDebugger.py.</td>
</tr>
<tr>
<td>2019-07-16</td>
<td>Added semihosting input and debuggable state support to Python/Examples/IrisDebugger.py.</td>
</tr>
<tr>
<td>2019-07-23</td>
<td>Added Python/Examples/IrisViewer.py, an experimental Iris low-level debugger.</td>
</tr>
<tr>
<td>2019-08-01</td>
<td>Added function instance_getCppInterfaceIrisInstance() to allow enhancing IrisInstances with tightly coupled plugins.</td>
</tr>
<tr>
<td>2019-08-02</td>
<td>Made cnname fields mandatory (in ResourceInfo and ResourceGroupInfo), according to the specification.</td>
</tr>
<tr>
<td>2019-09-26</td>
<td>Added Examples/Plugin/EnhanceInstance which shows how to enhance a C++ IrisInstance with a tightly coupled plugin.</td>
</tr>
</tbody>
</table>
Chapter 2
Iris examples

This chapter describes the examples of Iris clients, plug-ins, and Python scripts. Each client and plug-in example contains source code and a makefile for GCC or project file for Microsoft Visual Studio.

It contains the following sections:
• 2.1 C++ client examples on page 2-19.
• 2.2 DummyModel example on page 2-20.
• 2.3 Plug-in examples on page 2-21.
• 2.4 Python examples on page 2-22.
2.1 C++ client examples

The following C++ client examples are located in $IRIS_HOME/Examples/Client/.

--- Note ---

- When launching the model to connect to, use the -I option to start an Iris server.
- Many of these examples require you to specify a target instance. You must use the full hierarchical instance name, for example:

  ```
  ./breakpoints localhost:7100 component.FVP_Base_Cortex_A32x1.cluster0.cpu0 --list
  ```

For information about Iris instance names, see 5.20.1 Hierarchical instance names and instance classes on page 5-172.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoints</td>
<td>• Sets or deletes code or data breakpoints.</td>
</tr>
<tr>
<td></td>
<td>• Prints information about breakpoints that have been set for a target instance.</td>
</tr>
<tr>
<td>Connection</td>
<td>• Connects this client instance to the Iris server.</td>
</tr>
<tr>
<td></td>
<td>• Prints the client instance name and id.</td>
</tr>
<tr>
<td>Disassembly</td>
<td>• Prints the disassembly modes that are supported by a target instance.</td>
</tr>
<tr>
<td></td>
<td>• Prints the current disassembly mode.</td>
</tr>
<tr>
<td></td>
<td>• Disassembles a chunk of memory, or an individual opcode.</td>
</tr>
<tr>
<td>ExecutionControl</td>
<td>• Starts or stops the simulation.</td>
</tr>
<tr>
<td></td>
<td>• Halts or resumes execution of a specific instance.</td>
</tr>
<tr>
<td></td>
<td>• Provides an interactive mode, which in addition:</td>
</tr>
<tr>
<td></td>
<td>— Prints the execution time and execution state of the simulation.</td>
</tr>
<tr>
<td></td>
<td>— Performs instruction stepping.</td>
</tr>
<tr>
<td>Memory</td>
<td>• Prints information about the memory spaces that are exposed by the target instance.</td>
</tr>
<tr>
<td></td>
<td>• Prints the contents of memory that was read from a specific address, in a specific memory space.</td>
</tr>
<tr>
<td>Register</td>
<td>• Prints information about all resource groups and the resources (registers and parameters) in each group, for a target instance.</td>
</tr>
<tr>
<td></td>
<td>• Writes a value to a specific resource.</td>
</tr>
<tr>
<td></td>
<td>• Reads a value from a specific resource.</td>
</tr>
<tr>
<td>Semihosting</td>
<td>• Creates an event stream for IRIS_SEMIHOSTING_OUTPUT events.</td>
</tr>
<tr>
<td></td>
<td>• Registers itself to receive callbacks for semihosting output events.</td>
</tr>
<tr>
<td></td>
<td>• Receives semihosting output from the target and prints it.</td>
</tr>
<tr>
<td>SimpleTraceClient</td>
<td>• Registers itself to receive a specific trace source from all instances. The trace source is specified on the command line, or INST by default.</td>
</tr>
<tr>
<td></td>
<td>• Creates an event stream for the trace source specified.</td>
</tr>
<tr>
<td></td>
<td>• Implements a callback function that prints the contents of each trace event.</td>
</tr>
</tbody>
</table>

---

2 Iris examples
2.1 C++ client examples
2.2 DummyModel example

 DummyModel is an example of a complete Iris system, including the main server-side components that are required by Iris. Its purpose is only to demonstrate Iris, all other functionality in the example can be ignored.

 DummyModel is aimed at:
 • Simulator developers. It shows how to integrate Iris support into a simulation framework.
 • Component developers. It shows how to use Iris to expose aspects of a component, for example registers, parameters, memory, and event sources.

 It provides a simple Iris framework, so can also be used to test new Iris-related features and reproduce problems outside of a full simulation framework.

 It consists of the following:

 **Simulation engine**
   Its main purpose is to maintain an event queue for Iris calls.

 **Main executable**
   Its main purpose is to instantiate components and load plug-in libraries.

 **Component**
   Uses Iris to expose aspects of a component.

 This example takes the following command-line options:

 -p <port_number>       Set the TCP server port. The default is the first free port in the range 7100-7109.
 -P                      Print the port number that the Iris server is listening to.
 -G                      Print a log of all Iris messages. This option can be specified multiple times, for example -G -G for log level 2.
 -S                      Enable verbose logging in the IrisTcpServer. This option can be specified multiple times, for example -S -S for log level 2.
 -A                      Allow remote connections from another machine to the Iris server. Defaults to not allowed.
 --list-params           Print parameters and exit.
 --instantiate=[0,1]     If 0, do not automatically instantiate the simulation. This allows a remote client to connect to and instantiate it. Defaults to 1.
 --plugin <plugin_name>  Load a plug-in library.
 -C <param>=<value>      Set a parameter.
## 2.3 Plug-in examples

The plug-in examples are located in `$IRIS_HOME/Examples/Plugin/`.

The following examples are DSO plug-ins that you can build, then load into a model using the `--plugin` option.

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EnhanceInstance</strong></td>
<td>A plug-in that is tightly coupled with another instance. The plug-in uses the C++ interface of the target instance to add some registers to it.</td>
</tr>
</tbody>
</table>
| **GenericTrace** | • Iris implementation of the GenericTrace MTI plug-in.  
|                  | • Prints a comma-separated list of trace sources specified on the command line (INST by default) to a file or to stdout.  
|                  | • Specify the trace sources of interest, and optionally the trace file to write to, as parameters to the plug-in. Each parameter is prefixed with GenericTrace, for example:    
|                  | `./isim_system --plugin $IRIS_HOME/.../GenericTrace.so -C GenericTrace.event=EXCEPTION,EXCEPTION_RETURN` |
| **ListInstances** | • Registers to receive IRIS_SIM_PHASE_END_OF_ELABORATION events.  
|                 | • At the end of elaboration, a callback prints to stdout details of all registered instances in the simulation. |
| **ListResources** | Configurable plug-in that can print to a file or to stdout any of the following information, in JSON format, for each instance in the simulation:  
|                   | • Instance name, id, and properties.  
|                   | • Event sources supported by the instance.  
|                   | • Parameters and registers that the instance exposes.  
|                   | Additionally, it can print information about the simulator and hidden data.  
|                   | Configure the plug-in using parameters. To see the list of plug-in parameters, run the model with the `-l` option. The plug-in parameters are prefixed with ListResources.  
|                   | For example, to output register information, use:  
|                   | `./isim_system --plugin $IRIS_HOME/.../ListResources.so -C ListResources.registers=1` |
| **SimPhaseEvents** | • Registers callback functions for IRIS_SIM_PHASE_* events.  
|                  | • Each callback prints the name of the event and the time it occurred. |
| **SimpleTrace** | Similar to the GenericTrace plug-in example, except it only traces a single trace source (INST by default) and prints to stdout only. Use the SimpleTrace.event parameter to specify the trace source, for example:  
|                | `./isim_system --plugin $IRIS_HOME/.../SimpleTrace.so -C SimpleTrace.event=CORE_REGS` |
2.4  Python examples

The following examples demonstrate the `iris.debug` Python module. They are located in `$IRIS_HOME/Python/Examples/`.

For more information about `iris.debug`, see *Iris Python Debug Scripting User Guide*.

--- Note ---

- Before running these scripts, you must have added the directory that contains `iris.debug` to the `PYTHONPATH` environment variable.
- When launching the model to connect to, use the `-I` option to start an Iris server.
- Some of these examples load an application into the model. The application must be located in the directory from which the model was launched, in other words, the current working directory.

---

### Table 2-3  Python examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectModel.py</td>
<td>• Connects to a running model.</td>
</tr>
<tr>
<td></td>
<td>• Prints the names of all target component instances.</td>
</tr>
<tr>
<td></td>
<td>• Prints the CPU instance properties.</td>
</tr>
<tr>
<td>DemoBreakpoints.py</td>
<td>• Loads an application (<code>vectors.axf</code>) into the model.</td>
</tr>
<tr>
<td></td>
<td>• Sets register, program, and memory breakpoints.</td>
</tr>
<tr>
<td></td>
<td>• Runs the model and prints information about the breakpoints that were hit.</td>
</tr>
<tr>
<td>DemoControl.py</td>
<td>Connects to the model, then runs, stops, and steps it.</td>
</tr>
<tr>
<td>DemoMemory.py</td>
<td>• Loads an application (<code>endian.axf</code>) into the model.</td>
</tr>
<tr>
<td></td>
<td>• Writes data to memory then reads it back as bytes and words, to show the effect of changing endianness.</td>
</tr>
<tr>
<td>DemoParameters.py</td>
<td>• Prints all the parameters for the first CPU in the model.</td>
</tr>
<tr>
<td></td>
<td>• Changes a run-time parameter and prints the new value.</td>
</tr>
<tr>
<td>DemoRegisters.py</td>
<td>Modifies registers in the CPU in the target model and prints the values before and after the changes.</td>
</tr>
<tr>
<td>LoadApplication.py</td>
<td>• Loads an application (<code>demo.axf</code>) into the model.</td>
</tr>
<tr>
<td></td>
<td>• Steps through the program.</td>
</tr>
<tr>
<td></td>
<td>• Modifies and prints some register values.</td>
</tr>
<tr>
<td>SemiHostConf.py</td>
<td>• Loads an application (<code>semihostconf.axf</code>) into the model.</td>
</tr>
<tr>
<td></td>
<td>• Handles semihosted I/O by printing the semihosting stderr and stdout of the CPU target to the console.</td>
</tr>
</tbody>
</table>
Chapter 3
Generic function call interface

Iris interfaces are named functions that receive named arguments. The functions mostly receive and return structured data as objects that contain named values. In case of an error, they return an error code. This chapter describes how to access Iris interfaces from C++ or using IPC/TCP.

It contains the following sections:
• 3.1 JSON data types on page 3-24.
• 3.2 JSON-RPC 2.0 function call format on page 3-26.
• 3.3 Synchronous and asynchronous behavior on page 3-28.
• 3.4 Sending a request, a notification, and a response on page 3-29.
• 3.5 U64JSON on page 3-31.
• 3.6 Function call optimizations on page 3-39.
• 3.7 IrisC interface on page 3-40.
• 3.8 IrisRpc (RPC transport layer) on page 3-48.
• 3.9 JSON-RPC 2.0 over HTTP on page 3-52.
• 3.10 Threading model and ordering on page 3-53.
3.1 JSON data types

Iris interfaces use JSON data types. The JSON type system provides clarity and simplicity and is supported by all relevant programming languages.

In this topic, Value represents any of the following JSON types or constants:

- **Object.** A map from String to Value.
- **Array.** A list of Values, not necessarily of the same type.
- **String.**
- **Number.** Represents integer and floating point values of arbitrary precision. Iris avoids using the arbitrary Number type.
- **Boolean.**
- **True.**
- **False.**
- **Null.**

The term NumberU64 refers to a `uint64_t`, in the range 0 to \(2^{64}-1\).

The term NumberS64 refers to an `int64_t`, in the range \(-2^{63}\) to \(2^{63}-1\).

The term `Type[]` refers to an array of `Type` values. For example String[] is an array of strings. Value[] is semantically identical to Array.

The term Map[String]Type refers to a map or dictionary-like object where the type of the key is String and the type of the value is `Type`. For example, Map[String]NumberU64 is a map or dictionary object with String keys and NumberU64 values. Map keys are always Strings. Map[String]Value is semantically identical to Object.

The following table defines the implicit type conversions that are performed on the interface boundary. The first column contains the values to convert from:

<table>
<thead>
<tr>
<th>Original value</th>
<th>To Number</th>
<th>To NumberU64</th>
<th>To NumberS64</th>
<th>To Boolean</th>
<th>To Object</th>
<th>To Array</th>
<th>To String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>✓</td>
<td>-/✓ (Round)(Range)</td>
<td>-/✓ (Round)(Range)</td>
<td>-/✓ (Round)(only 1/0)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NumberU64</td>
<td>✓</td>
<td>✓</td>
<td>-/✓ (Range)</td>
<td>-/✓ (only 1/0)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>NumberS64</td>
<td>✓</td>
<td>-/✓ (Range)</td>
<td>✓</td>
<td>-/✓ (only 1/0)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Boolean</td>
<td>✓ (to 1/0)</td>
<td>✓ (to 1/0)</td>
<td>✓ (to 1/0)</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Object</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Array</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>String</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Key:**

✓ Implicitly converted.
- Not converted. `E_*_type_mismatch` error.

(Round) Floating point numbers are implicitly rounded to the nearest integer.

(Range) Implicitly converted when in range, else `E_*_type_mismatch` error.

(only 1/0) Numbers are converted to Booleans when they are 1 (True) or 0 (False), else `E_*_type_mismatch` error, which is stricter than C, C++, and Python.

(to 1/0) Booleans are implicitly converted to numbers as 1 (True) or 0 (False), as in C, C++, and Python.
Note

- The Iris interface only expects and produces U64 and S64 numbers. Other numbers can be passed to Iris functions and are converted according to this table.
- Null is not implicitly converted to or from anything else.
3.2 JSON-RPC 2.0 function call format

Iris interfaces use the JSON-RPC 2.0 format and semantics for function calls and responses. JSON-RPC 2.0 is a lightweight Remote Procedure Call (RPC) mechanism that is easy to understand and implement.

In this documentation, the terms request, notification, and response refer to the Request, Notification, and Response Objects that are defined by JSON-RPC 2.0, see http://www.jsonrpc.org/specification.

All functions are called by name, and responses are associated with requests by using a request id, which the caller assigns to the request.

The JSON-RPC 2.0 standard supports various use patterns. For example, it supports argument passing by position or by name. Iris uses the following subset of, and extensions to, JSON-RPC 2.0:

Function arguments
Function arguments are passed by name, not by position. In other words, params is an object, not an array. The order of the arguments in the Iris API documentation is irrelevant.

When manually generating JSON or U64JSON requests, you can order function arguments alphabetically by name to speed up function call processing.

Request ids
The request id that is passed to a function call is a NumberU64. The caller specifies bits[31:0] of the request id. This allows the caller to match function return values with function calls. It is usually an integer that increases with every request. This increasing id can wrap around, but ids of ongoing function calls must not be reused. There is no requirement to use an increasing id or even unique ids.

The instId argument specifies the instance that the function operates on, similar to a this pointer in C++. For more information about instId, see 4.2 Instances on page 4-57. The caller must set bits[63:32] of the request id to the instance id, instId, of the caller. This part of the id is used to route responses back to the caller. The usage of bits[63:32] is an Iris-specific extension of JSON RPC 2.0 to support the Iris Object Model. For more information about the Object Model, see Chapter 4 Object model on page 4-55

Note
Notifications do not need to route a response back to the caller. Therefore, notifications do not use a request id.

Requests and notifications
All Iris functions can be sent either as a request or as a notification, unless otherwise stated in the API documentation:
• For requests, the caller always receives a response, even if the function is specified to have no return value. In this case, a Null value is returned.
• For notifications, the caller never receives a response and no value is returned, even if the function is specified to return one.

String encoding
All strings, including object member names, are encoded using UTF-8. All Iris function names, function argument names, and object member names are plain ASCII and are C identifiers. Iris does not use String to transport binary data, because String cannot represent all binary byte sequences. For example, Strings cannot contain NUL bytes.

Binary data
Binary data is transported as a NumberU64[] with an explicit size argument, if necessary.
Case sensitivity
The following are case sensitive:
• Function names.
• Argument names.
• Object member names.
• Instance names.
• Event source names.
• Event source field names.
• Any other names and textual identifiers.
• Type strings.
• Verbatim strings that are used in the interface.

Bi-directional calls
Functions can generally be called in both directions between two instances.

Batch requests
Iris does not support batch requests or batch responses, in other words, arrays of requests or arrays of responses. Independent function calls to independent instances do not map well onto the batch requests and batch responses as defined in JSON-RPC 2.0. However, IrisRpc uses persistent TCP connections, and sending individual requests has almost the same performance as sending them in an array as a batch request.
3.3 Synchronous and asynchronous behavior

When calling a function, you can generally choose either to generate a synchronous request, which will later be answered by the callee with a response, or you can generate an asynchronous notification for which no response is generated.

Requests enable the caller to know when the callee has finished with the function, while notifications do not. This is relevant in the following use-cases:

**Synchronous or asynchronous event callbacks**
When enabling an event callback, the event consumer can choose:

- Whether the event-generating instance should be blocked while the consumer processes the event. This is a synchronous callback, using a request and a response for each event, see `syncEc=True` argument to `eventStream_create()`.
- Whether the event-generating instance should continue running, without waiting for the event to be processed. This is an asynchronous callback, using a notification.

**Overlapping function calls**
In general it is unnecessary to wait for a response before issuing the next request, except for instances that generate an event callback when `syncEc=True`. This means that function calls can overlap.

For example, if a debugger wants to read all registers, some memory, and a table of information, it issues the following requests without waiting for them to complete:

- `resource_read()`, request id=707.
- `memory_read()`, request id=708.
- `table_read()`, request id=709.

It then waits for the responses to requests 707, 708, and 709 in any order. This reduces the round-trip latency, for example through a TCP connection, from three round trips to one round trip, but makes the code more complex.

**Related references**
5.17.7 `eventStream_create()` on page 5-150
3.4 Sending a request, a notification, and a response

This topic describes typical sequences of steps involved in sending a request, a notification, and a response.

Sending a request

Requests are defined in the JSON RPC 2.0 specification. For more information, see 3.2 JSON-RPC 2.0 function call format on page 3-26.

A request sent by the caller to the callee, and a response sent by the callee to the caller, are equivalent to a function call with a return value. The caller typically takes the following steps to complete a blocking function call:

Note

Before carrying out these steps, the caller must know its own instId, by calling instanceRegistry_registerInstance(). Typically, the caller also must have called instanceRegistry_getList() to find out the id of the instance it is calling.

1. The caller chooses a 32-bit request id, which it will use to match responses to requests. It puts its own instId, which is also 32 bits wide, into the top 32 bits of the request id to form the 64-bit request id that is passed with the request. This is required by the global instance to route responses back to the caller.
2. The caller encodes the JSON RPC 2.0 request object in U64JSON.
3. The caller calls irisHandleMessage() on the IrisInterface that it is connected to. This is usually provided by IrisCoreConnection for in-process instances or IrisClientConnection for out-of-process instances. This forwards the request to the callee.
4. The caller can do other work or send other requests to the same instance or to other instances.
5. The caller receives the response, which comes through the irisHandleMessage() function of the IrisInterface of the caller.

In practice, these steps are handled by a support library and the caller is not exposed to them. Requests can be sent from different threads. The caller does not have to wait for a response before sending another request. Requests can overlap. Responses can come from different threads and in any order. The caller must use the request id to match the response to a request.

Sending a notification

Notifications are defined in the JSON RPC 2.0 specification. For more information, see 3.2 JSON-RPC 2.0 function call format on page 3-26. Notifications are used, for example, by non-blocking, asynchronous callbacks. Sending a notification differs from sending a request in the following ways:

• The caller does not specify a request id. This also means that the caller does not send its instance id to the callee.
• The caller does not receive any response, including any error response, not even E_function_not_supported_by_instance, E_unknown_instance_id, or any low-level I/O error codes from the transport layer. This might limit the usefulness of notifications.
• The callee does not send a response.

Sending a response

Responses are defined in the JSON RPC 2.0 specification. For more information, see 3.2 JSON-RPC 2.0 function call format on page 3-26. After processing the request, the callee takes the following steps to send a response to the request:

1. Extracts the instId of the caller from the request id of the request.
2. Constructs a JSON RPC 2.0 response object encoded in U64JSON using the caller's instId and the original 64-bit request id.

Related references

5.20.12 instanceRegistry_registerInstance() on page 5-179
5.20.11 instanceRegistry_getList() on page 5-179
3.5 U64JSON

This section defines the Iris-specific binary variant of JSON called U64JSON.

U64JSON uses a sequence of uint64_t values to represent JSON data. It is fully equivalent to JSON and can be converted into JSON and back without data loss. The U64JSON variant is used whenever in-process communication takes place and it can optionally be used over IPC.

The main motivation for U64JSON is fast generation and consumption of arbitrary structured data, function calls, and return values, especially in-process.

This section contains the following subsections:
- 3.5.1 U64JSON format on page 3-31.
- 3.5.2 Container length on page 3-36.
- 3.5.3 Endianness on page 3-37.
- 3.5.4 Signedness and integer representation on page 3-37.
- 3.5.5 Numbers with arbitrary size and precision on page 3-37.
- 3.5.6 Optimizations and normalized form on page 3-37.
- 3.5.7 U64JSON examples on page 3-38.

3.5.1 U64JSON format

In U64JSON, each JSON value is encoded as a sequence of uint64_t values.

The following terminology is used in the table below:

**MSB**

The most significant 4 or 8 bits of each uint64_t value, in other words, bits[63:60] or bits[63:56]. The MSB determines the type and encoding of the value.

**Container length**

The number of uint64_t values, including the leading MSB value, representing the Value. This is the number of uint64_t values that must be skipped when skipping the Value. If the container length is not specified in the table, it is one.

**Array length**

The number of elements in an array. This is not the same as the container length.

**Value**

Any JSON value that is encoded according to this table.

---

**Note**

For examples of U64JSON, see 3.5.7 U64JSON examples on page 3-38.

---

<table>
<thead>
<tr>
<th>MSB</th>
<th>JSON type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Number</td>
<td>Positive integer Numbers from 0 to 0xffffffffffffff (60 bits).</td>
</tr>
<tr>
<td>0x1</td>
<td>Number</td>
<td>Negative Numbers from -0x1000000000000000 to -1. To convert such a negative 64-bit pattern to U64JSON, bits[63:60] (0xf) are replaced with 0x1. To convert the U64JSON representation back to a 64-bit pattern, bits[63:60] (0x1) are replaced with 0xf.</td>
</tr>
</tbody>
</table>
### Table 3-2 U64JSON format (continued)

<table>
<thead>
<tr>
<th>MSB</th>
<th>JSON type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 0x2-0x7 | String | String. Short string, 255 or fewer bytes long, where string[6] is in 0x20-0x7f if longer than 6 bytes. Format:  
```c
// nn = string length, s_±MM = string
uint64_t msb_and_string_data = 0xMMss ssss ssss sssn;
uint64_t more_string_data_if_necessary[n >> 3]; // missing for nn <= 7
```
If the string is less than 7 bytes long, bits[63:56] (MM) are set to 0x20. This makes the MSB 0x2-0x7. String bytes are stored in little-endian format. All padding bytes are 0 bytes, except for the 0x20 in MM for short strings <= 6 bytes.
The string can contain null bytes and is not zero-terminated. The string encoding is UTF-8.

**Note**
- All ASCII strings that contain only printable characters, and therefore all C identifiers, which are 255 or fewer bytes long, belong to this class. This includes all Iris function names, argument names, and object member names, and also a lot of transported strings, like resource names.
- Strings with 15 or fewer bytes can be compared with one or two uint64_t compares.
- Encoding and decoding on little-endian machines is efficient (string follows length byte).

Container length: (nn >> 3) + 1.

| 0x8 | NumberU64[] | Array of NumberU64 values. Format:  
```c
uint64_t msb_and_array_length = 0x8nnn nnnn nnnn nnnn; // n_ = array length
uint64_t data[n];
```
Data is not encoded according to this table but rather stored as plain uint64_t values.
Container length: n_ + 1.

| 0x9 | - | Reserved. |

| 0xa | Array | Generic array which can contain anything. Format:  
```c
// x_ = container length
uint64_t msb_and_container_length = 0xaxxx xxxx xxxx xxxx;
uint64_t array_length;
Value elements[array_length];
```
Container length: x_. |
<table>
<thead>
<tr>
<th>MSB</th>
<th>JSON type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 0xb  | Object    | Object container. Map from String to Value. Format:

```c
// x... = container length
uint64_t msb_and_container_length = 0xbxxx xxxx xxxx xxxx;
uint64_t number_of_members;
struct { String member_name; Value value; } members[number_of_members];
```

Container length: x....

When converting from JSON, the object members should be sorted alphabetically by member name. This neither restricts nor enhances JSON, because in JSON, object members have no defined order. When converting to JSON, the object members can be emitted in alphabetical order or in any other order.

The reason for ordering object members is that U64JSON is used for time-critical, in-process function calls. Function arguments can be found faster in an ordered list than in an unordered list.

The reason for using alphabetical ordering is that function calls can be converted to and from JSON and the arguments are represented as an object, in which no order can be relied upon. Alphabetical order can be mechanically re-established, regardless of the Object's semantics.

| 0xc0 | Number   | 64-bit NumberU64. Only used if the number cannot be represented using the MSB4=0 or MSB4=0xf formats. Format:

```c
uint64_t msb = 0xc000 0000 0000 0000;
uint64_t number;
```

Container length: 2.

| 0xc1 | Number   | 64-bit NumberS64. Only used if the number cannot be represented using the MSB4=0, MSB4=0x1, or MSB8=0xc0 formats. Format:

```c
uint64_t msb = 0xc100 0000 0000 0000;
uint64_t number;
```

Container length: 2.

| 0xc2-0xc9 | -       | Reserved, except 0xc9fffffffffffffffff is an invalid U64JSON encoding and causes an E_u64json_encoding_error. Implementations can use this internally to explicitly represent an invalid U64JSON value, without affecting future extensions.

| 0xca    | Number   | 64-bit double-precision floating-point number. Format:

```c
uint64_t msb = 0xca00 0000 0000 0000;
double number;
```

Container length: 2.

| 0xcb    | -       | Reserved. |
Table 3-2  U64JSON format (continued)

<table>
<thead>
<tr>
<th>MSB</th>
<th>JSON type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 0xcc  | String    | String. All strings that are not represented using the MSB=0x2-0x7 format, that is, either of the following:  
|       |           | • Strings that are 256 or more bytes long.  
|       |           | • Strings that are 7 or more bytes long and string[6] not in 0x20-0x7f.  
|       |           | Format:  
|       |           | \[
|       |           | uint64_t msb_and_string_length = 0xccnn nnnn nnnn nnnn; // n... = string length  
|       |           | uint64_t string_data[(n + 7) >> 3]  
|       |           | String bytes are stored in little-endian format. Unused bytes, if any, must be set to zero. The string can contain null-bytes and is not zero-terminated. The string encoding is UTF-8.  
|       |           | Container length: (n...+15) >> 3. |
| 0xcd  | Null      | Null value. Format:  
|       |           | uint64_t msb = 0xcd00 0000 0000 0000; |
| 0xce  | Boolean   | Boolean value False. Format:  
|       |           | uint64_t msb = 0xce00 0000 0000 0000; |
| 0xcf  | Boolean   | Boolean value True. Format:  
|       |           | uint64_t msb = 0xcf00 0000 0000 0000; |
| 0 xd  | -         | Reserved |
## Table 3-2 U64JSON format (continued)

<table>
<thead>
<tr>
<th>MSB</th>
<th>JSON type</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| 0xe0 | Request   | Encodes an Iris request message.  

```c
uint64_t msb = 0xe0vv xxxx xxxx xxxx xxxx;
```

`vv` indicates the JSON-RPC version and is 0x20.  

`x...` is the container length.

```c
uint64_t request_id; // request[1]
```

This is the same as the `id` field in a JSON-encoded request. The `request_id` is a plain `uint64` and is not encoded according to the Number format in this table.

```c
uint64_t instId; // request[2]
```

Indicates the destination instance, in other words, the callee of the Iris function. Equivalent to the `instId` parameter that is required by almost all Iris functions. Setting `instId` to zero is equivalent to omitting the `instId` parameter, targeting the global instance. The `instId` is a plain `uint64` and is not encoded according to the Number format in this table.

```c
String method; // request[3:n]
```

Equivalent to the `method` field in a JSON-encoded request. This field has a variable length and is encoded as a U64JSON string according to the format in this table. The offset `n` is given by the encoding of the string.

```c
Object params; // request[n:m]
```

Equivalent to the `params` field in a JSON-encoded request. This field has a variable length and is encoded as a U64JSON object according to the format in this table. The offset `n` is given by the encoding of the previous field and `m` by the encoding of the `params` object.

| 0xe1 | Notification | Encodes an Iris notification message.  

The encoding of a Notification is the same as that of a Request except for the following differences:

- The MSB is 0xe1 instead of 0xe0.
- The `request_id` field is ignored and should be set to 0xffffffffffffffff. All other fields are encoded in the same way as a Request.
### MSB | JSON type | Meaning
--- | --- | ---
0xe2 | Response | Encodes an Iris response message.

```c
uint64_t msb = 0xe2vv xxxx xxxx xxxx xxxx;
```

**vv** indicates the JSON-RPC version and is 0x20.

x... is the container length.

```c
uint64_t request_id; // response[1]
```

This is the same as the id field in a JSON-encoded response. The `request_id` is a plain `uint64_t` and is not encoded according to the Number format in this table.

```c
uint64_t instId; // response[2]
```

Indicates the destination instance, in other words, the caller of the Iris function. The `instId` is a plain `uint64_t` and is not encoded according to the Number format in this table.

```c
int64_t error_code; // response[3]
```

If this is zero (E_ok), this response returns a result. Any other value is an Iris error code and this response returns an error. The value of this field determines the encoding of the rest of the Response. The error code is a plain `int64_t` and is not encoded according to the Number format in this table. See Chapter 6 Response error codes on page 6-202 for information about error codes.

```c
Value result; // response[4:n]
```

or

```c
String message; // response[4:m]
```

The type and meaning of this field depends on the value of the preceding `error_code` field. If `error_code` is E_ok, this field is the result of the call encoded as a U64JSON value. It can have any type. If `error_code` is not E_ok, this field is the response error object `message` field encoded as a U64JSON string.

```c
Value data; // response[m:xx]
```

Optional error object `data` field. This field is only present if this is an error response, in other words, `error_code` is not E_ok.

This can be any value encoded as U64JSON according to this table. The data field is optional and can be omitted. In this case, the Response container ends at the end of the `message` field and \( m == xx \).

<table>
<thead>
<tr>
<th>MSB</th>
<th>JSON type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf</td>
<td>Number</td>
<td>Positive integer Numbers from 0xf000000000000000-0xxffffffffffffffff (60 bits).</td>
</tr>
</tbody>
</table>

### 3.5.2 Container length

The type and total length of each JSON Value can be determined by looking at its first `uint64_t` value. This means that any structured type, whether an object, array, or string, can be skipped in constant time because it does not need to be parsed. This enables access to any object member in linear or constant time.

For objects and the array variant 0xa, the container length is explicit and is redundant. For all other value types, it is implicit.
3.5.3 Endianness

When used in-process, the `uint64_t` array is transported with the native endianness of each `uint64_t` value.

--- Note ---

Values larger than `uint64_t` are created by using a sequence of `uint64_t` values in little-endian format, with the lowest `uint64_t` value first, independent of the host endianness. However, this is out of scope of this documentation.

When used over IPC, for instance over TCP, the `uint64_t` array is serialized in little-endian format.

3.5.4 Signedness and integer representation

JSON can unambiguously represent positive and negative integers of arbitrary size. U64JSON can represent all integers in the range \(-2^{63}\) to \(2^{64}-1\) unambiguously.

Almost all Iris interfaces specify the signedness of each integer value, so it is clear whether it is a signed or unsigned integer. This means that programming languages can use any 64-bit data type to represent the 64-bit patterns transported through JSON or U64JSON. Applications must make sure that these 64-bit patterns are then suitably interpreted when processed further. There are very few interfaces in which integer values are allowed to cover the whole signed and unsigned range, for example in parameters. It is only necessary in these few cases to support signed and unsigned 64-bit integers at the same time. This can be achieved by storing an explicit type flag, for example `bool isSigned` in addition to the 64-bit pattern.

3.5.5 Numbers with arbitrary size and precision

JSON can represent numbers with arbitrary size and precision. U64JSON can only represent signed and unsigned 64-bit integers and 64-bit double-precision floating point values.

Interfaces that support larger numbers or bit patterns must represent them using JSON values that can hold an arbitrary amount of data. For example, `NumberU64[]` is used by the resources and memory interfaces to represent arbitrarily wide bit patterns.

3.5.6 Optimizations and normalized form

Some JSON values can be represented in more than one way in U64JSON.

- Numbers must be represented according to this list, with highest priority first:
  - **Small (60 bits) positive integers in the range 0 to \(2^{60}-1\)**
    MSB is `0x0`.
  - **Small (60 bits) negative integers in the range \(-2^{60}\) to \(-1\)**
    MSB is `0x1`.
  - **Small (60 bits) positive integers in the range \(2^{64}\) to \(2^{64}-1\)** (`0xF000000000000000` to `0xFFFFFFFFFFFFFFFF`)
    MSB is `0xf`.
  - **Other positive 64-bit integers, in the range \(2^{60}\) to \(2^{64}-2^{60}\)** (`0x1000000000000000` to `0xEFFFFFFFFFFFFFFF`)
    MSB is `0xc0`.
  - **Other negative 64-bit integers, in the range \(-2^{63}\) to \(-2^{60}\)**
    MSB is `0xc1`.
  - **All floating-point numbers that can be represented by a double**
    MSB is `0xca`.
- Strings must be represented as follows:
Strings <= 255 bytes that have a 0x20-0x7f byte in s[6] if >= 6 bytes
MSB is 0x2-0x7.

All other strings
MSB is 0xcc.

• Arrays must be represented as follows:
  
  **Array of uint64_t**
  MSB is 0x8.

**Generic array**
MSB is 0xa.

• U64JSON messages must be represented as a Request, Notification, or Response, and never as an Object. This is important for efficient routing and decoding. Any message that is encoded as an Object receives an E_malformatted_request response.
• For Object, Boolean, and Null, there is only one possible representation.

Interface functions, their arguments, and return data are defined in terms of JSON in this documentation, not U64JSON. The U64JSON encoding is a fully equivalent alternative for encoding JSON data.

3.5.7 **U64JSON examples**

The following table gives some examples of JSON values encoded in U64JSON.

**Table 3-3  U64JSON examples**

<table>
<thead>
<tr>
<th>JSON value</th>
<th>U64JSON representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3]</td>
<td>0x8000000000000003, 1, 2, 3</td>
</tr>
<tr>
<td>[&quot;1&quot;,2,3]</td>
<td>0xa000000000000005, 3, 0x2000 0000 0000 3101, 2, 3</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>0x2000 0000 6362 6103</td>
</tr>
<tr>
<td>&quot;&quot;</td>
<td>0x2000000000000000</td>
</tr>
<tr>
<td>&quot;numbyte&quot;</td>
<td>0x6574 7962 6d75 6e07</td>
</tr>
<tr>
<td>&quot;numbytes&quot;</td>
<td>0x6574 7962 6d75 6e08, 0x73</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0xffff ffff ffff ffff</td>
<td>0xffff ffff ffff ffff</td>
</tr>
<tr>
<td></td>
<td>This is (2^{64}-1), not -1.</td>
</tr>
<tr>
<td>-1</td>
<td>0x1fff ffff ffff ffff</td>
</tr>
<tr>
<td>0xaabb ccdd eeff 0011</td>
<td>0xc000 0000 0000 0000, 0xaabb ccdd eeff 0011</td>
</tr>
<tr>
<td>-0x1234 5678 9012 3456</td>
<td>0xc100 0000 0000 0000, 0xedcb a987 6fed cbaa</td>
</tr>
<tr>
<td>{&quot;num&quot;:1,&quot;b&quot;:2,&quot;c&quot;:3}</td>
<td>0xb000 0000 0000 0000, 3, 0x2000 0000 6d75 6e03, 1, 0x2000 0000 0000 6201, 2, 0x2000 0000 0000 6301, 3</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong></td>
</tr>
<tr>
<td></td>
<td>13 JSON tokens (26 chars) translate into 8 uint64_t values (64 bytes) in U64JSON.</td>
</tr>
<tr>
<td>Null</td>
<td>0xcd00000000000000</td>
</tr>
<tr>
<td>False</td>
<td>0xce00000000000000</td>
</tr>
<tr>
<td>True</td>
<td>0xcf00000000000000</td>
</tr>
</tbody>
</table>
3.6 Function call optimizations

This section describes some optimizations that implementations can use to call functions more efficiently.

This section contains the following subsections:
- 3.6.1 Fast argument parsing using sorted arguments on page 3-39.
- 3.6.2 String comparison and hashing on page 3-39.

3.6.1 Fast argument parsing using sorted arguments

The Iris function call mechanism passes function arguments by name. It also passes object members, for example members of passed or returned structs, by name.

Therefore, it is possible for the caller to order arguments or members differently in each call. If so, the callee must repeatedly search the list of arguments or members. This has a large performance impact, particularly because functions might accumulate many optional arguments over time.

To avoid this problem, the caller should list the arguments for a function in alphabetically ascending order and the callee should parse the arguments in this order. If the caller and callee follow this rule then the argument list only needs to be parsed once. This does not impose any overhead on the caller.

However, the callee cannot always rely on a sorted argument list because this rule is not compulsory. The callee must support unsorted or incorrectly sorted lists.

3.6.2 String comparison and hashing

When an instance receives an incoming function call, in other words a request, it must first look up the function by name.

In U64JSON, all strings are sequences of 64-bit values. They have several properties that implementations can exploit to make function lookup more efficient:
- The first 64-bit value contains the first 7 characters and the length of the string.
- Most strings in Iris can be encoded using 1-6 uint64_t values, and most Iris function names can be encoded using 1-4 uint64_t values. It is possible to write explicit code for these four cases instead of using a generic loop.
- If the first 64-bit values of two strings are different, the two strings are different.
- If the first 64-bit values of two strings are the same, they are guaranteed not to be a prefix of each other, or the two strings are the same.

It is often possible to implement a function lookup that does not check for unknown functions in constant time by using closed hashing on the first 64-bit value and comparison of the second 64-bit value only if necessary. Checking for unknown functions can be a runtime option which then uses a slower decoder (debug mode).
3.7 IrisC interface

IrisC is the low-level C interface that is used between shared libraries in an Iris system. Typically, instances do not deal with IrisC directly but use a support library, such as IrisSupportLib, to provide a high-level abstraction.

This section contains the following subsections:
• 3.7.1 Memory and interface ownership on page 3-40.
• 3.7.2 IrisCore lifecycle functions on page 3-40.
• 3.7.3 IrisClient lifecycle functions on page 3-40.
• 3.7.4 General IrisC functions on page 3-41.
• 3.7.5 Plug-in API on page 3-44.

3.7.1 Memory and interface ownership

Function pointers and context pointers that are passed to an IrisC function are owned by the instance that originated them and must stay valid for the lifetime of the instance.

All other memory that is passed to an IrisC function, for example a U64JSON-encoded message that is passed as a `uint64_t` pointer, is owned by the caller and must not be accessed by the callee after the call has returned. The callee must make a copy of memory if it needs to access it later.

3.7.2 IrisCore lifecycle functions

IrisSupportLib defines the following functions to manage its lifecycle:

**IrisCore_init()**

```
int64_t IrisCore_init(void **iris_core_context_out)
```

Initialises the GlobalInstance and provides an IrisC context pointer that should be used for all future calls to IrisC functions.

Arguments:

- `iris_core_context_out` Output argument. The value of `*iris_core_context_out` is set to the IrisCore context pointer.

Return value:

An IrisErrorCode value indicating whether the call was successful.

**IrisCore_shutdown()**

```
int64_t IrisCore_shutdown(void *iris_core_context)
```

Destroys the GlobalInstance and shuts down Iris. All instances are unregistered and any running server is shut down. The context pointer passed in should not be used after this function returns.

Arguments:

- `iris_core_context` Context pointer returned by `IrisCore_init()`.

Return value:

An IrisErrorCode value indicating whether the call was successful.

3.7.3 IrisClient lifecycle functions

IrisSupportLib defines the following functions to manage its lifecycle:
**IrisClient_connect()**

```c
int64_t IrisClient_connect(void **iris_client_context_out, const char *hostname, uint16_t port);
```

Initialises an IrisTcpClient and connects it to an Iris server.

**Arguments:**

- **iris_client_context_out**
  - Output argument. The value of *iris_client_context_out* is set to the IrisClient context pointer that the server was successfully connected to.

- **hostname**
  - Hostname of the server to connect to.

- **port**
  - Server port to connect to.

**Return value:**

An IrisErrorCode value indicating whether the call was successful.

**IrisClient_disconnect()**

```c
int64_t IrisClient_disconnect(void *iris_client_context);
```

Disconnects and destroys an IrisTcpClient. If a client disconnects from the server spontaneously it should still call IrisClient_disconnect() to clean up any state allocated by the IrisTcpClient.

**Arguments:**

- **iris_client_context**
  - Context pointer returned by IrisClient.

**Return value:**

An IrisErrorCode value indicating whether the call was successful.

### 3.7.4 General IrisC functions

These IrisC functions are implemented by IrisSupportLib and some must also be implemented by users of IrisSupportLib.

**handleMessage()**

```c
int64_t handleMessage(void *handle_message_context, const uint64_t *message);
int64_t IrisCore_handleMessage(void *iris_core_context, const uint64_t *message);
int64_t IrisClient_handleMessage(void *iris_client_context, const uint64_t *message);
```

Passes a message to be routed or handled. IrisCore and IrisClient define handleMessage() functions to route a message to its destination instance by calling the handleMessage() function for that instance. If handleMessage() returns E_ok, this does not imply that a request was handled successfully or even that it has been handled at all. The response to a request is delivered by calling the handleMessage() function for the caller with a response message.

**Arguments:**

- **handle_message_context**
  - The context pointer for the callee. For IrisCore_handleMessage(), this is the iris_core_context pointer provided by IrisCore_init(). For IrisClient_handleMessage(), this is the iris_client_context pointer provided by IrisClient_connect(). For other handleMessage() functions, this is the context pointer associated with that function.
message
A U64JSON-encoded message. This can be a request, a notification, or a response.

Return value:
An IrisErrorCode value indicating whether the call was successful.

registerChannel()

`int64_t IrisCore_registerChannel(void *iris_core_context, IrisC_CommunicationChannel *channel, uint64_t *channel_id);`
`int64_t IrisClient_registerChannel(void *iris_client_context, IrisC_CommunicationChannel *channel, uint64_t *channel_id);`

In order for IrisCore and IrisClient to route messages to an instance they must know the handleMessage() function and context pointer to use to pass a message to that instance. These pointers are grouped together into the IrisC_CommunicationChannel structure and registered with the IrisC library by calling registerChannel().

Arguments:

**iris_core_context or iris_client_context**
The context pointer for the callee. For IrisCore_registerChannel(), this is the iris_core_context pointer provided by IrisCore_init(). For IrisClient_registerChannel(), this is the iris_client_context pointer provided by IrisClient_connect().

**channel**
An IrisC_CommunicationChannel struct for the channel being registered.

**channel_id**
Output argument. *channel_id is set to an id number used by IrisCore or IrisClient to identify the channel. This id is used when registering instances using the instanceRegistry_registerInstance() Iris function.

Return value:
An IrisErrorCode value indicating whether the call was successful.

unregisterChannel()

`int64_t IrisCore_unregisterChannel(void *iris_core_context, uint64_t channel_id);`
`int64_t IrisClient_unregisterChannel(void *iris_core_context, uint64_t channel_id);`

Unregisters a previously registered channel when an instance disconnects from the Iris system.

Arguments:

**iris_core_context or iris_client_context**
The context pointer for the callee. For IrisCore_unregisterChannel(), this is the iris_core_context pointer provided by IrisCore_init(). For IrisClient_unregisterChannel(), this is the iris_client_context pointer provided by IrisClient_connect().

**channel_id**
The id for the channel being unregistered. Any instances that have been registered using this channel are automatically unregistered if they have not already done so themselves.

Return value:
An IrisErrorCode value indicating whether the call was successful.
IrisC_CommunicationChannel structure

IrisC_CommunicationChannel structure.

```c
struct IrisC_CommunicationChannel {
    uint64_t CommunicationChannel_version;
    IrisC_HandleMessageFunction handleMessage_function;
    void *handleMessage_context;
};
```

Members:

**CommunicationChannel_version**

This member must be set to 0.

**handleMessage_function**

`handleMessage()` function pointer for this channel.

**handleMessage_context**

Context pointer to pass when calling `handleMessage_function`.

**processAsyncMessages()**

`processAsyncMessages()` function.

```c
int64_t IrisCore_processAsyncMessages(void *iris_core_context, uint64_t flags);
int64_t IrisClient_processAsyncMessages(void *iris_client_context, uint64_t flags);
```

Processes any buffered messages for the current thread. If an instance is using thread marshalling to ensure that all messages are handled on the same thread, that instance must ensure that `processAsyncMessages()` is being called from that thread to forward marshalled messages.

Arguments:

**iris_core_context or iris_client_context**

The context pointer for the callee. For `IrisCore_processAsyncMessages()`, this is the `iris_core_context` pointer provided by `IrisCore_init()`. For `IrisClient_processAsyncMessages()`, this is the `iris_client_context` pointer provided by `IrisClient_connect()`.

**flags**

Bitwise OR of IrisC_AsyncMessage flags.

Return value:

An IrisErrorCode value indicating whether the call was successful.

**IrisC_AsyncMessage flags**

IrisC_AsyncMessage flags.

**IrisC_AsyncMessage_Default**

Default non-blocking behavior. Returns immediately if there are no outstanding messages to process.

**IrisC_AsyncMessage_Blocking**

If there are no outstanding messages to process, block until there is one. This is useful when waiting for a response to a request. Call `processAsyncMessages()` as follows to wait for the response and also to avoid deadlock situations in which the recipient of your request makes a request that needs to be marshalled to the thread being blocked:

```c
while (!no_response_received)
    processAsyncMessages(context, IrisC_AsyncMessage_Blocking);
```
irisInitPlugin()

irisInitPlugin() function.

```c
int64_t irisInitPlugin(IrisC_Functions *functions);
```

Iris plug-in entry point. This function should be exported by an Iris plug-in DSO.

Arguments:

**functions**
A pointer to an IrisC_Functions struct that contains pointers to IrisC functions so that the plug-in can register instances and interact with Iris.

Return value:
An error code indicating whether the call was successful.

**IrisC_Functions structure**

IrisC_Functions structure.

```c
struct IrisC_Functions {
    uint64_t Functions_version;
    void *iris_c_context;
    IrisC_RegisterChannelFunction registerChannel_function;
    IrisC_UnregisterChannelFunction unregisterChannel_function;
    IrisC_HandleMessageFunction handleMessage_function;
    IrisC_ProcessAsyncMessagesFunction processAsyncMessages_function;
};
```

Members:

**Functions_version**
This member must be set to 0.

**iris_c_context**
Context pointer to use when calling all IrisC functions.

**registerChannel_function**
Pointer to an IrisC library registerChannel() function.

**unregisterChannel_function**
Pointer to an IrisC library unregisterChannel() function.

**handleMessage_function**
Pointer to an IrisC library handleMessage() function.

**processAsyncMessages_function**
Pointer to an IrisC library processAsyncMessages() function.

### 3.7.5 Plug-in API

Plug-ins can play any role in a system, for example, a debugger, a client, a visualization tool, a trace receiver, a trace generator, a component, a part of a component, or any combination of these.

The plug-in API consists of the following:

- **irisInitPlugin()** entry point. This allows the plug-in to make and receive Iris function calls.
- Iris function calls, made in both directions.
- Iris initialization phase callbacks. These callbacks have names beginning IRIS_SIM_PHASE_. They allow the plug-in to hook into the initialization and shutdown processes.

$\$IRIS_HOME/Examples/Plugin/ contains source code for some example Iris plug-ins.
Initialization phase callbacks

Initialization phase callbacks allow plug-ins, and any other instances, to hook into the initialization and shutdown stages. Depending on its role, a plug-in might perform initialization in different callbacks, in order to provide information to other parts of the system as early as possible.

The initialization phase callbacks are Iris events, without fields.

——— Note ———
For an example plug-in that registers callback functions for IRIS_SIM_PHASE_* events, see $IRIS_HOME/Examples/Plugin/SimPhaseEvents/.

———

This is the list of callbacks, in the order in which they are called on an instance:

**IRIS_SIM_PHASE_INITIAL_PLUGIN_LOADING_COMPLETE**
Called just after all plug-ins have been loaded on simulation startup. This is the earliest point in time at which all plug-ins can discover the presence of all other initial plug-in instances, because plug-ins register themselves as at least one instance in irisInitDso().

——— Note ———
Plug-ins can be loaded and unloaded dynamically afterwards. This callback is only called once, after the initial plug-in loading has completed.

**IRIS_SIM_PHASE_INSTANTIATE_ENTER**
Called just before **IRIS_SIM_PHASE_INSTANTIATE**. No component instances have been created yet.

**IRIS_SIM_PHASE_INSTANTIATE**
Called as part of the system instantiation phase. This is when all component instances are created. Plug-ins can use this step to emulate instantiating themselves at the same time as components.

**IRIS_SIM_PHASE_INSTANTIATE_LEAVE**
Called just after **IRIS_SIM_PHASE_INSTANTIATE**, which is just after all component instances have been instantiated. All component instances are usually connected, but are not yet initialized.

**IRIS_SIM_PHASE_INIT_ENTER**
Called just before **IRIS_SIM_PHASE_INIT**. Connections to other components are already established, but other components are not yet initialized.

**IRIS_SIM_PHASE_INIT**
Called as part of the **init()** phase of all components. A component typically initializes itself here. Other components might or might not be initialized yet.

**IRIS_SIM_PHASE_INIT_LEAVE**
Called just after **IRIS_SIM_PHASE_INIT**, which is after **init()** of all components. Other components are already initialized, in the sense of **init()**. This is the earliest point when all trace sources of all components can be discovered.

**IRIS_SIM_PHASE_BEFORE_END_OF_ELABORATION**
Called just after **IRIS_SIM_PHASE_INIT_LEAVE**. In SystemC contexts, this is called in before_end_of_elaboration(). This is also called in non-SystemC contexts.

**IRIS_SIM_PHASE_END_OF_ELABORATION**
Called just after **IRIS_SIM_PHASE_BEFORE_END_OF_ELABORATION**. In SystemC contexts, this is called in end_of_elaboration(). This is also called in non-SystemC contexts.

**IRIS_SIM_PHASE_INITIAL_RESET_ENTER**
Called just before **IRIS_SIM_INITIAL_PHASE_RESET**.
**IRIS_SIM_PHASE_INITIAL_RESET**

Called as part of the the first `reset()` phase of components. This is only called once, after `init()`.

**IRIS_SIM_PHASE_INITIAL_RESET_LEAVE**

Called just after `IRIS_SIM_PHASE_INITIAL_RESET`.

**IRIS_SIM_PHASE_START_OF_SIMULATION**

Called just after `IRIS_SIM_PHASE_INITIAL_RESET_LEAVE`. In SystemC contexts, this is called in `start_of_simulation()`. This is also called in non-SystemC contexts.

**IRIS_SIM_PHASE_RESET_ENTER**

Called just before `IRIS_SIM_PHASE_RESET`.

**IRIS_SIM_PHASE_RESET**

Called as part of the the `reset()` phase of components. This is called for every simulation reset, not hardware reset, after the first one. See `IRIS_SIM_PHASE_INITIAL_RESET` for the first invocation of `reset()` after `init()`. To achieve the semantics of component `reset()`, combine `IRIS_SIM_PHASE_INITIAL_RESET` and `IRIS_SIM_PHASE_RESET`.

**IRIS_SIM_PHASE_RESET_LEAVE**

Called just after `IRIS_SIM_PHASE_RESET`.

**IRIS_SIM_PHASE_END_OF_SIMULATION**

Called just before `IRIS_SIM_PHASE_TERMINATE_ENTER`. In SystemC contexts, this is called in `end_of_simulation()`. This is also called in non-SystemC contexts.

**IRIS_SIM_PHASE_TERMINATE_ENTER**

Called just before `IRIS_SIM_PHASE_TERMINATE`. This is the last chance to access components before the `terminate()` phase is called on them. This is generally the last chance to access components in a safe way.

**IRIS_SIM_PHASE_TERMINATE**

Called as part of the `terminate()` phase of components.

**IRIS_SIM_PHASE_TERMINATE_LEAVE**

Called just after `IRIS_SIM_PHASE_TERMINATE`. As components might already have freed their resources, it is not safe to access other components from this callback.

**Plug-ins and callbacks**

Different types of plug-in must do work in different callbacks.

**Trace plug-ins and debugger plug-ins**

To receive trace events from components, a trace plug-in typically does work in the following callbacks:

**IRIS_SIM_PHASE_INIT_LEAVE**

This corresponds to the MTI `registerSimulation()` callback. The plug-in can discover all trace sources here.

**IRIS_SIM_PHASE_INITIAL_RESET_LEAVE**

Here, all components are properly initialized and their register values are properly reset.

**IRIS_SIM_PHASE_TERMINATE_ENTER**

This is the last chance to read the final trace state of components.

Trace plug-ins should generally not do any work in the following callbacks because it is unclear which part of the observed components have completed these stages and which have not:

- `IRIS_SIM_PHASE_INSTANTIATE`.
- `IRIS_SIM_PHASE_INIT`.
- `IRIS_SIM_PHASE_RESET`.
- `IRIS_SIM_PHASE_TERMINATE`.
To provide debugger-like functionality or to simply observe the state of components, a plug-in typically does work in the same callbacks as trace plug-ins.

**Special plug-ins**

A plug-in can discover other plug-ins that were loaded at startup in `IRIS_SIM_PHASE_INITIAL_PLUGIN_LOADING_COMPLETE`, for example to print a list of all plug-in instances.

---

**Note**

The behavior of a plug-in should not depend on other plug-ins because this violates user expectations. Plug-ins should not influence each other, unless this influence is their main purpose.

---

**SystemC simulation phases**

The Iris initialization phase callbacks occur during the following SystemC simulation phases:

<table>
<thead>
<tr>
<th>Iris initialization phase callback</th>
<th>SystemC simulation phase</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IRIS_SIM_PHASE_INITIAL_PLUGIN_LOADING_COMPLETE</code></td>
<td><code>sc_main()</code>, before <code>sc_start()</code></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INSTANTIATE_ENTER</code></td>
<td><code>before_end_of_elaboration()</code></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INSTANTIATE</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INSTANTIATE_LEAVE</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INIT_ENTER</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INIT</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INIT_LEAVE</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_BEFORE_END_OF_ELABORATION</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_END_OF_ELABORATION</code></td>
<td><code>end_of_elaboration()</code></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INITIAL_RESET_ENTER</code></td>
<td><code>start_of_simulation()</code></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INITIAL_RESET</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_INITIAL_RESET_LEAVE</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_START_OF_SIMULATION</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_RESET_ENTER</code></td>
<td><code>sc_start()</code>, after <code>start_of_simulation()</code>, and before <code>end_of_simulation()</code></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_RESET</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_RESET_LEAVE</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_END_OF_SIMULATION</code></td>
<td><code>end_of_simulation()</code></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_TERMINATE_ENTER</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_TERMINATE</code></td>
<td></td>
</tr>
<tr>
<td><code>IRIS_SIM_PHASE_TERMINATE_LEAVE</code></td>
<td></td>
</tr>
</tbody>
</table>
3.8 IrisRpc (RPC transport layer)

IrisRpc is the low-level protocol that allows Iris clients to connect to Iris servers and to exchange IrisRpc messages, which are used to make Iris function calls.

IrisRpc assumes a bi-directional byte stream between the client and the server. This section assumes a TCP connection is used, but any other bi-directional byte stream transport can be used instead, for example Unix pipes, or a serial line.

The version of the IrisRpc transport protocol that is described in this section is 1.0. This does not indicate an Iris interface version or a level of support for Iris functions. Support for Iris interfaces can be queried using `instance_checkFunctionSupport()` on page 5-180.

In this section, the term *client* means an instance, for example a debugger, connecting to a running server, and *server* refers to, for example, the IrisTcpServer. The server represents the simulation executable. Multiple clients can connect to a server at any time.

The JSON RPC 2.0 specification uses the terms *client* and *server* to indicate the caller and callee, respectively. These semantics are not used in this documentation. An Iris client is both a JSON RPC 2.0 client and a server. An Iris server is also both a JSON RPC 2.0 client and server. Both clients and servers can send and receive functions calls.

---

**Note**

- The client and server side must both implement a JSON RPC 2.0 client and server. This is mandatory. This means both sides can call functions on the other side.

  Some functions in this documentation are called *callbacks*. This term refers to a function that is called in the other direction in a specific context. Callbacks are normal function calls.

- All interactive clients should be able to receive callbacks. To receive callbacks, clients must use a persistent connection.

- When a client disconnects, all its callbacks are automatically unregistered. The IrisTcpServer discards callbacks that should be sent to a disconnected client. This might happen if unregistering the callback was delayed.

- Killing the model and killing a client are first-class operations and must be supported seamlessly.

---

This section contains the following subsections:

- 3.8.1 IrisRpc connection handshake on page 3-48.
- 3.8.2 Rejecting a connection request on page 3-49.
- 3.8.3 Supported formats on page 3-49.
- 3.8.4 IrisRpc message format on page 3-50.
- 3.8.5 TCP considerations on page 3-50.

### 3.8.1 IrisRpc connection handshake

Clients use the IrisRpc protocol to initiate a connection to an Iris server, for example the IrisTcpServer that is running in the simulation.

The IrisRpc connection handshake allows the client to do the following:

- Verify that the server it is connecting to is an Iris server.
- Request a specific IrisRpc version.

It also allows the server to notify the client whether it supports the requested IrisRpc version.

For a TCP connection, it is assumed that the client knows the host IP and port number of the TCP server.

The following procedure is used to establish a connection:

---

**Note**

<CR> and <LF> represent ASCII 13 and 10 respectively.
• The client connects to the server, for example it connects to the TCP port.
• The client sends the following request to connect using IrisRpc version 1.0:

```
CONNECT / IrisRpc/1.0<CR><LF>
Supported-Formats: IrisJson, IrisU64Json, JsonRpcOverHttp<CR><LF>
```<CR><LF>

• The server sends the following response to tell the client that the connection is established, the protocol is IrisRpc/1.0, and a list of supported message formats:

```
IrisRpc/1.0 200 OK<CR><LF>
Supported-Formats: IrisJson, IrisU64Json, JsonRpcOverHttp<CR><LF>
```<CR><LF>

• After this step, the client and server can send and receive IrisRpc messages.

At this point, all client instances, but usually just one, should call `instanceRegistry_registerInstance()` to register themselves in the instance registry. Other instances can then discover them and query their properties and name.
• The server keeps the connection open until the simulation executable terminates. The client keeps the connection open until it either terminates or it no longer needs the connection to the server.

Note
Closing the connection implicitly unregisters all client instances that used this connection from the instance registry, destroys all event streams, and unregisters all other callbacks and artefacts.

The start lines and the header fields must be formatted according to RFC 7230, Section 3, HTTP/1.1 Message Format. The client and the server must ignore any header fields they do not understand.

**Related references**
5.20.12 `instanceRegistry_registerInstance()` on page 5-179

### 3.8.2 Rejecting a connection request

A client might reject a server, or a server might reject a client, for the following reasons:

• If a client requests an IrisRpc protocol version that the server does not support, the server can send the following response after the CONNECT step:

```
IrisRpc/0.1 505 IrisRpc version not supported<CR><LF>
Error-Message: This IrisTcpServer only supports IrisRpc/0.x.<CR><LF>
```<CR><LF>

The server, then the client, closes the connection.

• A server might reject a client because it does not support any of the formats that the client supports. The server responds with:

```
IrisRpc/0.1 501 Format not supported<CR><LF>
Error-Message: This IrisTcpServer only supports the formats IrisU64Json, IrisJson but the client does not support any of these.<CR><LF>
```<CR><LF>

• A client might reject a server because it does not support any of the formats that the server supports. In this case, the client closes the connection without any further communication with the server.

Note
The last two cases can only occur when the client was configured to force a format other than IrisJson, because all servers and clients support the IrisJson format.

### 3.8.3 Supported formats

The `Supported-Formats:` header in the CONNECT request or response can contain the following case-sensitive values.

Supporting a format means both for sending and receiving.
IrisJson
IrisRpc protocol using JSON format.

IrisU64Json
IrisRpc protocol using U64JSON format.

JsonRpcOverHttp
JSON-RPC over HTTP. This format is initiated using a HTTP POST or GET request, not a CONNECT request. This format cannot be used after a CONNECT request.

All servers and clients must at least support the IrisJson format. Therefore, an incompatibility should not occur. Servers should support all of the formats listed.

3.8.4 IrisRpc message format

IrisRpc messages transport JSON RPC 2.0 function calls, responses, and notifications.

The IrisRpc message format provides the following functionality:

- Allows senders to send messages either in JSON, U64JSON, or another format.
- Allows receivers to detect whether a message is in JSON, U64JSON, or another format.
- Allows receivers to determine the length of a message, in order to efficiently read it without parsing the content of the message.
- Allows receivers to detect when they are out of sync, and re-sync.

IrisJson format:

```
IrisJson:<ascii_decimal_content_length>:<content><LF>
```

Where:

- `<ascii_decimal_content_length>` is the length of `<content>` in bytes as a decimal ASCII number. This must not contain any leading zeros and must be at most ten decimal digits.
- `<content>` is a JSON RPC 2.0-encoded function call, response, or notification.
- `<LF>` is a byte with the value of 10.

IrisU64Json format:

```
IrisU64Json:<uint32_le_content_length><content><LF>
```

Where:

- `<uint32_le_content_length>` is the length of `<content>` in bytes as a 32-bit little-endian unsigned integer.
- `<content>` is an array of little-endian encoded uint64_t values. It is a JSON RPC 2.0 and U64JSON-encoded request, response, or notification.

Senders must send messages in a format that is supported by the receiver. Senders know which formats are supported from the handshake. The format might change from message to message inside a session. An exception is connections that were initiated with JsonRpcOverHttp, which must use JsonRpcOverHttp by both sides for the session.

Receivers inspect the first few bytes of a received message to determine the format. When the format is unknown or not supported, they close the TCP connection immediately without reporting an error to the TCP peer.

3.8.5 TCP considerations

TCP sockets should be created with the keep-alive option, if possible. If not possible, the side that does not support keep-alive should call the instance_ping(instId=0) function, which does nothing, 3600s after the last function call. The default TCP keep-alive time is 7200s, or 2 hours, for Linux and Windows.

The two crossed JSON RPC 2.0 client and JSON RPC 2.0 server pairs share a single TCP connection. Function calls, responses, and notifications are sent over the same TCP connection in both directions.
Related references
5.20.18 instance_ping() on page 5-182
3.9 JSON-RPC 2.0 over HTTP

In addition to IrisRpc, Iris supports the HTTP standard transport for making functions calls. The IrisTcpServer supports it transparently.

Iris uses the specification http://www.simple-is-better.org/json-rpc/transport_http.html, although this section overrides some parts of the specification, in particular, persistent connections.

This section contains the following subsections:
- 3.9.1 Recognizing a session as JSON-RPC HTTP
- 3.9.2 Persistent connections

3.9.1 Recognizing a session as JSON-RPC HTTP

The IrisTcpServer recognizes an HTTP session by receiving a POST request line, instead of a CONNECT request line for IrisRpc. Clients do not have to do anything special.

- The IrisTcpServer only supports POST calls:
  - Content-Type: must be application/json-rpc.
  - Content-Length: must contain the correct length according to the HTTP specification.
  - Accept: must be application/json-rpc.
- The server responds to GET and PUT with 405 Method Not Allowed.
- The URL for POST is / . The URL is neither used to identify functionality nor to identify a specific target in the simulation, nor to pass parameters. Everything is specified inside the JSON-RPC message. The HTTP wrapper contains no semantic information.

The server reads header fields that only relate to establishing the connection for the first message, and ignores them for any subsequent messages.

3.9.2 Persistent connections

The IrisTcpServer only supports keep-alive mode, which keeps the TCP connection open, even after responding to a request. Clients must send Connection: keep-alive in the header in every request. Clients can close the TCP connection at any time to end the session.

The IrisTcpServer does not support the no keep-alive mode, because closing the TCP connection implies the client is disconnecting. If a client calls instanceRegistry_registerInstance() without keep-alive, it would receive a response and would be assigned an instance id, but when the client or server closes the TCP connection, the client would automatically be removed from the instance registry and therefore could not make any more Iris function calls.

The IrisTcpServer accepts and produces chunked transfer encoding messages to implement bi-directional Iris messages. The client can send Iris messages either as a sequence of POST requests, or as a sequence of chunks of an initial POST request, or any combination. The IrisTcpServer responds with a sequence of chunks, until it receives a new POST request from the client. An end-of-chunks marker carries no semantic information. Switching between chunks and POST requests has no meaning. Clients must accept chunked transfer encoding. The IrisTcpServer does not send HTTP requests, for example POST, to the client.

As an alternative to using chunked transfer encoding, web sockets can be used to create a persistent connection between the client and the server.

Long polling cannot be used because it implies that a new TCP connection is created by the client to receive future events.

Related references

5.20.12 instanceRegistry_registerInstance() on page 5-179
3.10 Threading model and ordering

This section describes how Iris handles asynchronous functions, and gives rules for using synchronous event callbacks.

This section contains the following subsections:

- 3.10.1 Asynchronous functions on page 3-53.
- 3.10.2 Reentrancy on page 3-53.
- 3.10.3 Ordering rules for requests, notifications, and responses on page 3-54.

3.10.1 Asynchronous functions

All Iris functions are asynchronous, unless stated otherwise in the function description.

Functions can be called either as a request or as a notification:

- Functions that are called as a request, receiving a response, might or might not have an effect by the time `irisHandleMessage(request)` returns. The function is guaranteed to have completed from the caller's viewpoint by the time the caller's `irisHandleMessage(response)` was entered. `irisHandleMessage(response)` can be called at any point after `irisHandleMessage(request)` was entered. `irisHandleMessage(response)` can be called before or after `irisHandleMessage(request)` returned.

- Functions that are called as a notification do not receive a response. They can take effect any time after `irisHandleMessage(notification)` was entered and before or after `irisHandleMessage(notification)` returns.

In general, `irisHandleMessage()` might be called from any host thread. An instance that set `marshalRequests=true` when registering itself is only called on the thread from which it issued the `instanceRegistry_registerInstance()` call.

Related references

5.20.12 instanceRegistry_registerInstance() on page 5-179

3.10.2 Reentrancy

`irisHandleMessage()` implementations must support reentrancy. It can be called, and therefore reentered, by any number of pair-wise different host threads at the same time.

- `irisHandleMessage()` can be called, and therefore reentered, by its own thread. The `irisHandleMessage()` implementation must support reentrancy by the same thread and must hold mutexes only for short periods, when recursive reentrancy cannot happen, for example only when modifying local data structures and not when forwarding calls, to avoid deadlocks.

- Recursive mutexes should not generally be used, because mutexes protect invariants and recursive mutexes do not.

- Calling Iris functions while the simulation is running, for example from asynchronous `ec_FOO()` callbacks, is allowed. Reading state, for example, using `resource_read()` or `memory_read()` is allowed, but the results are random because the model state is fluctuating. It would not be possible to retrieve a consistent state across two or more Iris function calls. By the time a client receives an asynchronous `ec_FOO()` callback, the model might have progressed, and any observed state is unlikely to be related to the event. A typical use case is to indicate progress to the user by reading and displaying an instruction count.

- Calling back the simulation from within synchronous `ec_FOO()` callbacks is much more restricted than calling back from asynchronous `ec_FOO()` callbacks because Iris calls cannot be scheduled, but must complete while the simulation is blocked in the instance generating the event. This should only be done in very specific circumstances. The documentation of the Events API describes what functionality can be expected from within synchronous `ec_FOO()` callbacks. Also, not all event sources support synchronous `ec_FOO()` calls.
Calling back into the simulation from within a synchronous `ec_FOO()` callback of instance A has the following implications:

— If there are multiple parallel simulation threads, reading state from other instances might or might not be synchronous, for example when reading state from a completely unrelated instance that is still progressing. In this case, the Iris call is scheduled onto the other thread and is effectively asynchronous.

— If there is only a single simulation thread, the whole simulation is blocked by the synchronous event callback. All state is stable, but the state of instance A and all related instances might be inconsistent, depending on the nature of the event.

— As a guideline, data that is directly related to the event should be taken from the event fields rather than being read directly from the instance. For example, if the event is notifying about a bus fault, the `ec_FOO()` callback should not try to read fault registers, or registers and memory that are related to the current transaction. Instead, it should interpret the fields transported with the event and it can read unrelated state, for example the PC registers of all cores.

— Instances indicate that they do not support certain functions, or accessing certain state, while they are blocked in a synchronous event, by returning `E_not_supported_while_instance_is_blocked`.

**Related references**

5.17.5 `ec_FOO()` on page 5-148  
5.7.15 `memory_read()` on page 5-96  
5.6.13 `resource_read()` on page 5-76

### 3.10.3 Ordering rules for requests, notifications, and responses

Notifications and responses are asynchronous and can be called from any host thread. A response might be received before or after `irisHandleMessage(request)` returns.

The following ordering rules apply to requests, notifications, and responses:

- All requests, notifications, and responses from instance A to instance B arrive at instance B in the same order they were sent by instance A, if the order of these events was defined in A at all. If events E1 and E2 are generated in A with no implicit or explicit order, for example from two simulation threads without explicit synchronization, then the order of E1 and E2 is also undefined in B.
- All requests from instance A to instance B are completed in order at instance B. This means for function calls F1 and then F2 from A to B that F2 only starts to have an effect on B after F1 has completed, in other words, after B has sent the response for F1. F1 and F2 do not run concurrently.
- When instance A receives a response for function call F1 from instance B, function F1 has completed in B.
- When instance B receives the global event `ec_FOO(IRIS_SIMULATION_TIME_EVENT, RUNNING=False)`, which causes the simulation time to stop, it can be sure that it has received all requests and notifications that were generated up to and including this event.

These ordering rules have the following effects:

- Sending a sequence of requests or notifications from one instance to another without waiting for a response executes the requests or notifications in the order in which they were sent.
- Sending requests from one instance to another and waiting for the response to each request before sending the next request executes all requests in the order they were called.
- If instances A1 and A2 send notifications, for example events, to instance B, there are no guarantees about the order in which B receives the events. However, B receives all events from A1 in the order that A1 generated them, and all events from A2 in the order that A2 generated them. Also, if a transaction travels through A1 and A2 and back again using a causal path, then all events generated on the way arrive at instance B in the same causal order. B implicitly serializes these causally-related events in the correct order, by queuing incoming `IrisInterface::irisHandleMessage()`s.

**Related references**

5.17.5 `ec_FOO()` on page 5-148
Chapter 4
Object model

Iris provides an object model in which all entities are represented by instances. Instances can discover other instances and can call functions on each other. For example, a debugger can read a register in a CPU model, and the CPU model can send trace data to the debugger. Both debugger and model are instances.

It contains the following sections:
• 4.1 Object model overview on page 4-56.
• 4.2 Instances on page 4-57.
4.1 Object model overview

Iris uses a very simple object model:

- A system consists of a set of instances.
- Each instance has a unique numeric instance id, `instId`.
- Each instance has a unique instance name. This also implies a hierarchy.
- Each instance registers itself in a global instance registry, which assigns it an instance id.
- Each instance can query the list of instances and can also be notified when new instances are registered or unregistered.
- Instances communicate with each other by specifying the instance id.
- There are two special instances, the GlobalInstance, which has instance id 0, and the SimulationEngine, which has instance id 1. The GlobalInstance contains the global instance registry.

The object model does not have a hierarchy, but the instance names imply a hierarchy. The hierarchical instance names assign each instance to a specific class by specifying the class as the top-level hierarchy level. For example: `component.mainboard.cluster0.cpu3` is of class `component`. The following classes are defined:

- `component`
- `client`
- `framework`

For more information, see 5.20.1 Hierarchical instance names and instance classes on page 5-172.
4.2 Instances

JSON RPC 2.0 is a procedural interface, not an object-oriented interface. However, Iris extends it so that functions can be called on specific instances.

It achieves this by using the following extensions:

**instId argument**

All instance-specific functions have an instId argument, which identifies the instance that the function operates on. This argument is always named instId. It is used by framework components to route requests and notifications to their destination. This is similar to the this pointer in C++ or the self argument in Python. Callers must first query the list of instances using instanceRegistry_getList() from the global instance, whose instId is zero, to get the id of another instance.

**Instance-specific request id**

The request id contains the instance id of the caller in bits[63:32]. The request id is a NumberU64, for all requests. It is used by framework components to route responses from the callee back to the caller.

All instances in a system, for example components, plug-ins, remote clients, and framework instances, can discover and communicate symmetrically with all other instances in the system.

All instances register themselves in a central instance registry, which assigns instance ids. See 5.20 Instance registry, instance discovery, and interface discovery API on page 5-172 for details about the instance registry.

Instances can implement a subset of, or even a superset of, the functions that are defined in the Iris APIs. Instances that do not support a specific function must return E_function_not_supported_by_instance for that function. See 5.20.5 Interface discovery on page 5-176 for more information.

Few Iris functions do not have an instId argument. These functions apply globally rather than to a single instance, for example, instanceRegistry_registerInstance(). However, most functions are instance-specific.

Global functions are implemented by the global instance, which has the pre-defined instance id of zero. Specifying an instId argument of zero is equivalent to specifying no instId argument at all. For more details, see 5.3 instId argument on page 5-62.

**Related references**

5.20.11 instanceRegistry_getList() on page 5-179
5.20.12 instanceRegistry_registerInstance() on page 5-179
Chapter 5
Iris APIs

This chapter describes the Iris APIs. It provides conceptual information for each API, followed by detailed reference information for each function, object, and event source in the API.

Functions can return a result or an error code. The function-specific error codes are listed for each function, although generic error codes, which any function can return, are not listed. All error codes are described in Chapter 6 Response error codes on page 6-202.

Note
This documentation uses the syntax `foo()` to refer to a function called `foo`. The trailing parentheses are not part of the function name. They are only used in the documentation to identify function names.

It contains the following sections:

- 5.1 Iris API documentation on page 5-60.
- 5.2 Naming conventions on page 5-61.
- 5.3 instId argument on page 5-62.
- 5.4 Compatibility rules for function callers and callees on page 5-63.
- 5.5 Iris-text-format on page 5-64.
- 5.6 Resources API on page 5-67.
- 5.7 Memory API on page 5-88.
- 5.8 Disassembly API on page 5-104.
- 5.9 Tables API on page 5-109.
- 5.10 Image loading and saving API on page 5-115.
- 5.11 Simulation time execution control API on page 5-123.
- 5.12 Debuggable state API on page 5-126.
- 5.13 Stepping API on page 5-130.
- 5.14 Per-instance execution control API on page 5-134.
• 5.15 Breakpoints API on page 5-136.
• 5.16 Notification and discovery of state changes API on page 5-144.
• 5.17 Events and trace API on page 5-145.
• 5.18 Semihosting API on page 5-161.
• 5.19 Simulation accuracy (sync levels) API on page 5-167.
• 5.20 Instance registry, instance discovery, and interface discovery API on page 5-172.
• 5.21 Simulation instantiation and discovery API on page 5-189.
• 5.22 Plug-in loading and instantiation API on page 5-197.
• 5.23 TCP server management API on page 5-199.
• 5.24 Checkpointing API on page 5-201.
5.1 Iris API documentation

This book uses the following conventions when referring to Iris functions and objects:

**Intuitive type names**

Objects that are used as arguments and return values have intuitive type names, for example `RegisterInfo`. These type names do not occur in the requests or responses themselves, but are used in the documentation to help to clarify the purpose and context of the data. They also define a name for derived interfaces like C++, which support type names.

**Return values**

Function calls in JSON RPC 2.0 either return a result or an error member in the response object. For each function, the documentation describes any Objects that it returns in the result and lists any function-specific error codes that it can return. All functions can also return one of the general error codes, which are not listed in the function documentation, for brevity.

**Function call parentheses**

The Iris documentation uses the syntax `foo()` to refer to a function called `foo`, although the trailing parentheses do not appear anywhere in JSON or in U64JSON-formatted function calls. The parentheses are used to intuitively identify function names. In practice, Iris functions are called by language bindings, for instance C++ or Python functions, which use the syntax with parentheses.
5.2  Naming conventions

This topic describes the naming conventions that Iris APIs use.

Table 5-1 Naming conventions for Iris functions, objects, and events

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Case</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function name</td>
<td>lowerCamelCase</td>
<td>instanceRegistry_registerInstance</td>
</tr>
<tr>
<td>Function argument name</td>
<td>lowerCamelCase</td>
<td>instId</td>
</tr>
<tr>
<td>Object member name</td>
<td>lowerCamelCase</td>
<td>bitWidth</td>
</tr>
<tr>
<td>Object type name</td>
<td>UpperCamelCase</td>
<td>RegisterInfo</td>
</tr>
<tr>
<td>Event name</td>
<td>UPPERCASE_WITH_UNDERSCORES</td>
<td>IRIS_BREAKPOINT_HIT</td>
</tr>
<tr>
<td>Event field name</td>
<td>UPPERCASE_WITH_UNDERSCORES</td>
<td>BPT_ID</td>
</tr>
<tr>
<td>-</td>
<td>lowercase_with_underscores</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

Acronyms are treated like normal words and are written in lowercase, but sometimes have an uppercase first letter, for example `isTcp`, `isCpp`, `isJson`.

**Function names**

Function names are hierarchical, with hierarchy levels separated by an underscore. All functions have at least two parts, namely the group or interface name, and the function name, for example `resource_read`. Function names might have more hierarchy levels to further group the functionality.

See also **5.20.8 Naming conventions for new functions** on page 5-177 for the naming conventions to use when enhancing the interface.

**Experimental functions**

Experimental functions are prefixed by `experimental_` to avoid namespace pollution. Experimental functions are not part of the official Iris interface. Their semantics, arguments, and return values can change without notice. Experimental functions might become part of the Iris interface, and then lose the prefix, or might be removed without replacement.

**Custom functions**

Custom functions are prefixed by `custom_companyName` to avoid name clashes when multiple companies extend the Iris interface with their own functions. Introducing custom functions must be avoided if possible. It is preferable to use a combination of registers, memory spaces, tables, and event sources instead.

**Function argument names**

Function argument names should be short, if possible less than 24 bytes long, so that function implementations can compare argument names with one or three `uint64_t` compares. Details about an argument can be put into a description string which is retrieved using `instance_getFunctionInfo()`, rather than in its name. However it is useful to indicate the units, for example bits, bytes, elements, or milliseconds, in the argument name if multiple interpretations are possible. For example, `tickHz` or `bitWidth`.

**Object member names**

Object member names, for example in return values or in complex arguments, should be less than 24 bytes long.
5.3 instId argument

An instId argument occurs in many different functions. In all cases, it defines the instance that a function call is sent to. It has a similar role to the this pointer in C++ and the self argument in Python.

For example, when the global instance receives the function call:

```python
func(name=“foo”, instId=42, value=-1, bar=[1, “2”, True])
```

it can infer that this function call must be sent to the instance with id 42, without knowing what `func()` does, or whether the instance supports `func()` at all.

The instId argument is used in a function-independent way by the following framework instances:

- The global instance uses instId to determine which connected component, plug-in, or Iris server it should route a function call to. It does this for all calls, no matter where they come from.
- The Iris server uses instId to select the connection, and therefore the client, that a function call should be sent to.
5.4 Compatibility rules for function callers and callees

Functions are called by name and function arguments are named, not positional. Function return values are often objects, or arrays of objects, that contain named values.

These principles allow Arm to enhance the interface in future without breaking compatibility. To achieve this, callers and callees must follow some rules.

Callers of functions must follow these rules:

- The argument list must contain all mandatory arguments.
- The argument list can contain any optional arguments.
- The argument list must not contain any arguments that are not listed in this document.
- The caller can rely on mandatory members in the return value objects.
- The caller must not rely on any optional members in the return value objects. If an optional return value member is missing, this must have the semantics described in this document.
- The caller must accept and ignore any unknown members in return value objects.

Callees, in other words function implementations, must follow these rules:

- If a mandatory argument is missing, an error must be returned.
- If an optional argument is missing, this must have the semantics described in this document.
- If an unknown argument is passed, an error must be returned.
- All mandatory members must be returned in the return value objects.
- Any set of optional members of return values can be returned in the return value objects.

These rules have the following implications:

- Callees can be enhanced to accept extra optional arguments.
- Callees can be enhanced to return extra return value members. Existing callers that are unaware of the new members will ignore them.
- Callers that rely on certain mandatory or optional arguments must reliably receive an error response if a new argument is not supported by the callee.
Some functions return format strings that allow clients to format and annotate data values into a compact string for display purposes. These format strings are in the Iris-text-format, which is described in this section.

This section contains the following subsections:
- **5.5.1 Format strings** on page 5-64.
- **5.5.2 References to variables** on page 5-64.
- **5.5.3 Conditional formatting** on page 5-66.

### 5.5.1 Format strings

Format strings consist of literal characters and references to variables. The set of defined variables is specified by the function that returns the format string and is out of scope of this section. For example, these variables might be the fields of an event or of a table record.

### 5.5.2 References to variables

References to variables generally have the following syntax:

```
%{varname[optional_bitrange]optional_format_spec}
```

_______ Note _______

The percentage sign %, braces {...}, and square brackets [...] shown here are literal characters. In the rest of this topic, square brackets are used to indicate optional components.

_________

**varname**

The set of defined variable names depends on the context in which the format string is returned. It might also be possible to access fields of sibling objects, in which case varname can contain dots as hierarchy level separators.

**optional_bitrange**

If this is included, the specified bits are extracted from the numeric variable, or the specified characters are extracted from a string variable. The syntax of optional_bitrange is:

```
range[.range]...
```

Where range is either:

**pos**

Extract a single bit at pos.

**msb:lsb**

Extract the range of bits from MSB to LSB, where msb >= lsb.

pos, msb, and lsb are positive decimal numbers. The dot is a literal character that separates multiple ranges.

**optional_format_spec**

The default is :x for numeric types and :s for strings. The syntax is:

```
:[width].[precision][format_char]
```

Or:

```
:(enum_spec[|enum_spec]...)
```
Where:

**width**
The minimum number of characters to be printed for a number or for a string. Unused leading characters are filled with zeros for `x` and `b` and with spaces for `d` and `u`. If `width` is less than zero, the value is left-adjusted and the rightmost characters are filled with spaces.

**precision**
For `e`, `f`, and `g`, the number of precision digits. For `s`, the maximum number of characters to print.

**format_char**
Specifies the format in which values are printed. It can be one of the following:

- `x` Hexadecimal, without the leading `0x`. The client decides whether to use uppercase or lowercase hex, the guideline is lowercase.
- `d` Signed decimal. Either a minus sign for negative numbers or no sign.
- `u` Unsigned decimal.
- `b` Binary.
- `e` Scientific notation.
- `f` and `g` Floating-point number. The value must be exactly 32 bits or 64 bits wide.
- `y` Exact symbol lookup. The value is looked up in the symbol table, with an exact match, but see the note following this list. If found, this is replaced by the symbol name. If not found, this is replaced by the hexadecimal value with a leading `0x`.
- `Y` Lower bound symbol lookup. The value is looked up in the symbol table, searching for the symbol that has the highest value that is less than or equal to the value of the specified symbol, see the note following this list. If found, which is usually the case, this is replaced by `symbol_name+offset_in_hex` or just `symbol_name` on an exact match. If not found, this is replaced by the hexadecimal value with a leading `0x`.
- `s` String. Can only be used for string types.

**Note**
Some contexts might require the lower bits of an address to be masked out, depending on the target instance and the symbol type, for example ignoring bit[0] for Arm cores.

**enum_spec**
Instead of displaying the numeric value, display literal text. The format of `enum_spec` is:

```
<text>[=number]
```

This allows you to specify dense or sparse enum symbols for numeric values. The counting of non-explicit enum numbers follows the C rules, starting at zero, and uses `last_number+1` if no number is specified.

To output a literal `%`, use two percentage signs, `%%`. In addition, all percentage signs that are not followed by either `{` or `[` are treated as a literal `%`. Enum strings cannot contain the `|` or `)` characters.

Errors might occur if names are undefined, or if objects return an error when being read. In this case, the reference should be replaced with `(error: <reason>)`.

**Examples**

```
%{mode} -> Display variable 'mode' in hex according to its bitwidth.
%{fifo_len:u} -> Display variable as unsigned decimal integer.
%{address:4x} -> Display address as hex integer with 4 digits for N < 2**16 and > 4 digits for bigger values
%{perm:(---|--x|--w|--wx)r--|--r--|--rx|rw--|rw-|rwx)} -> Display permissions as 'rwx' field.
%{status[7:0]} -> Display 2 hex digits for the 8-bit value taken from bits[7:0].
%{status[7:0.15:12]} -> Display 3 hex digits for the 12-bit value taken from bits[7:0] as msb and bits[15:12] as lsb
%{pc:y} -> Display symbol.
```
5.5.3 Conditional formatting

In some cases, the formatting of a variable might depend on the value of one or more bits in that variable or in another variable.

This can be expressed with the map statement:

```
%[map|varname[optional_bitrange]|default|key1=format1[|key2=format2]...]
```

The variable is replaced with either the default or any of the specified formats. It is replaced with the format that is defined for a specific numeric value (key) if the variable has this numeric value. If the variable has a value that is not listed in the mapping, then it is replaced with the default. default and all specified formats might in turn contain map statements and variable references. default and the formats might contain literal = characters.

For example, this statement prints either a 10-bit index or a 20-bit address in the status register, depending on bit[31] in the status register:

```
%[map|status[31]|0=index=status[11:2]|1=address=status[19:0]]
```
5.6 Resources API

A resource is either a parameter or a register. Parameters allow you to parameterize an instance, either at startup, these are called init-time parameters, or at run-time. Registers represent a piece of state of an instance that can be read and might be modifiable.

Clients first query a list of the available resources of an instance by calling `resource_getList()`.

In the Resources API, each resource is uniquely identified by an opaque resource id. The `resource_read()` and `resource_write()` functions accept lists of resource ids to allow efficient reading and writing of multiple resources.

Clients typically inspect the meta information of the `ResourceInfo` objects returned by `resource_getList()` and extract the resource ids of the resources they are interested in. These are the `ResourceInfo` fields that are typically used to discover or filter resources:

- **registerInfo**
  - If present, the resource is a register.

- **parameterInfo**
  - If present, the resource is a parameter.

- **name**
  - For registers, this is the architectural name.

- **canonicalRn**
  - This field is usually set for core CPU registers, for instance DWARF register numbers, see 5.6.7 ElfDwarf scheme for canonical register numbers on page 5-71.

- **tags.isPc** and other tags
  - These tags allow clients to understand the semantics of the most important registers found in CPU cores, for example PC, stack pointer, instruction counter.

To access resources by name or by `canonicalRn`, clients typically build maps from names to resource ids or from `canonicalRns` to resource ids, because the `resource_read()` and `resource_write()` functions only accept resource ids.

Clients can also query resource groups by calling `resource_getListOfResourceGroups()`. This function returns lists of resource ids that are suitable for reading or writing all resources of a group.

Target instances can expose zero or more resources. Target instances that expose no resources must either return `E_function_not_supported_by_instance` for all `resource_*()` functions or `resource_getList()` must return an empty list.

State that consists of smaller chunks that can be addressed and accessed in a uniform way should be represented as memory or a table instead of resources, see 5.7 Memory API on page 5-88 and 5.9 Tables API on page 5-109.

$IRIS_HOME/Examples/Client/Register/ contains an example client application that demonstrates how to use this API.

This section contains the following subsections:

- 5.6.1 Parameters and registers on page 5-68.
- 5.6.2 Resource groups on page 5-68.
- 5.6.3 Registers with fields on page 5-68.
- 5.6.4 Resource names on page 5-69.
- 5.6.5 Reading a resource on page 5-69.
- 5.6.6 Writing a resource on page 5-70.
- 5.6.7 ElfDwarf scheme for canonical register numbers on page 5-71.
- 5.6.8 ElfDwarf canonical register numbers on page 5-72.
- 5.6.9 Comparison of resource types on page 5-73.
- 5.6.10 Exposure of parameters on page 5-74.
5.6.11 resource_getList() on page 5-75.
5.6.12 resource_getListOfResourceGroups() on page 5-75.
5.6.13 resource_read() on page 5-76.
5.6.14 resource_write() on page 5-76.
5.6.15 ParameterInfo on page 5-77.
5.6.16 RegisterInfo on page 5-78.
5.6.17 ResourceGroupInfo on page 5-79.
5.6.18 ResourceInfo on page 5-80.
5.6.19 ResourceReadResult on page 5-84.
5.6.20 ResourceTags on page 5-85.
5.6.21 ResourceWriteResult on page 5-87.

5.6.1 Parameters and registers

A resource is either a parameter or a register, but not both.

**isParameter**

A resource is a parameter if and only if the parameterInfo object is present in the ResourceInfo object.

**isRegister**

A resource is a register if and only if the registerInfo object is present in the ResourceInfo object.

Components that implement a subset of the architectural registers should expose all of them, even if some are only implemented as stubs. The description of stub registers should indicate that they are stubs.

5.6.2 Resource groups

Every resource belongs to one or more resource groups. Resource group names are short, human-readable strings, for example GPR or FPU Shadow.

Query the list of resource groups using resource_getListOfResourceGroups(). Clients should display them as a flat list, in the same order as they appear in the array returned by resource_getListOfResourceGroups().

A resource group must contain either registers or parameters, but clients must handle gracefully any resource group that contains both.

All parameters of an instance belong to a single group named Parameters, so you can query all parameters of an instance by calling:

```plaintext
resource_getList(group = "Parameters")
```

5.6.3 Registers with fields

Iris enables parts of registers to be exposed as child registers. Parent and child registers are linked by the ResourceInfo.parentRscId field.

A register either has no parent, in which case it is called a top-level register, and its parentRscId is missing, or it has one parent register, in which case it is a child of that parent. Child registers can have their own child registers, although instances should not expose a register hierarchy that is more than one or two levels deep. Clients should display child registers below their parents, in the order they appear in the ResourceInfo array returned by resource_getList().

When reading or writing child registers, it is the responsibility of the instance to access the data in the correct location. The ResourceInfo might contain enough information to locate the child resource inside the parent, but clients can ignore this information and just present it as part of the description of a resource.
It is possible to represent parts of a parent register as a logical child register. Logical child registers have no lsbOffset, indicating that they are distributed across multiple, non-consecutive bit ranges inside their parent register. Logical child registers might also represent any other information that is related to their parent register. They can be a different type to their parent register and can be used to create a structure of registers underneath the register group level.

Clients can usually ignore the difference between logical child registers and non-logical child registers, but instance implementations might find it useful to expose this information.

--- Note ---

Parameters do not have child parameters.

### 5.6.4 Resource names

Parameter names must be unique within an instance. This means that clients can ignore the group name of a parameter. Clients must handle non-unique parameter names gracefully, for example by only using the first of the conflicting parameter names in the list returned by `resources_getList()`.

Register names of top-level registers, in other words registers without a parent, must be unique within a register group of an instance. Register names of child registers must be unique within their parent. Clients must handle violations of these rules gracefully.

Resource names (ResourceInfo.name) must only consist of printable ASCII characters, in the range `0x20-0x7e`. They can contain spaces, dots, and characters that are usually used as separators. ResourceInfo.cname must be a valid C identifier, that is, it must start with a letter or underscore and must consist of letters, digits, and underscores.

Resource group names must not conflict with resource names.

In scripts and other non-GUI contexts, it is often necessary to uniquely specify a resource. To do this, clients should ensure the following:

- All name matching must be case-sensitive.
- Hierarchical resource names are built by concatenating the individual components of the name, separated by dots.
- To make a resource name unique, it can optionally have a resource group name prepended to it, separated by a dot. If no group name is specified and the resource name is not unique, clients must access the first matching resource in the array returned by `resource_getList()`.
- Child resources require all parent resources up to the top-level resource to be prepended, separated by dots.

For example, to access a child register C in register MMU_Status-Flags in register group Control 0.0.1, a script can use either of the following names:

- Control_0_0_1.MMU_Status_Flags.C
- MMU_Status_Flags.C

### 5.6.5 Reading a resource

The semantics of reading a resource are peek rather than architectural read. If possible, all instance implementations should try to achieve side-effect free reads. This is required for non-intrusive debug.

- The `resource_read()` function supports reporting bits in registers with an undefined value. It is optional for instances to support this.
- It is not always possible to return the exact value of a resource, for example when the instance that exposes the resource is not in a debuggable state. The `resource_read()` function supports reporting approximate values.
- The `resource_read()` function does not fail with an error for existing resources that could not be read. Instead, such read errors are reported in the error member of `ResourceReadResult`, for each resource.
Examples

resource_read() returns one ResourceReadResult object, which contains the read result for all resources that were read, comprising values and errors. Numeric resource values and string resource values are split into two different arrays. Numeric resource values occupy one or more NumberU64 values in the array, depending on their width. For example:

- Reading 3 32-bit resources with the values 1, 2, and 0xf00daaaa respectively:

  ResourceReadResult.data = [1, 2, 0xf00daaaa]

- Reading one 8-bit, one 64-bit, and one 32-bit resource, with values 1, 2, and 3 respectively:

  ResourceReadResult.data = [1, 2, 3]

- Reading one 132-bit wide resource which has the value 0x9_ffeedccc_bbba9988_77665544_33221100:

  ResourceReadResult.data = [0x7766554433221100, 0xffeedccbbaa9988, 9]

- Reading a string resource containing the string "abc":

  ResourceReadResult.data = []
  ResourceReadResult.strings = ["abc"]

- Reading a 32-bit resource with value 1, a string resource with value "abc", and a 32-bit resource with value 3:

  ResourceReadResult.data = [1, 3]
  ResourceReadResult.strings = ["abc"]

Note

The following resource_read() operations give the same result as this example:

- Reading a string resource with value "abc", a 32-bit resource with value 1, and a 32-bit resource with value 3.
- Reading a 32-bit resource with value 1, a 32-bit resource with value 3, and a string resource with value "abc".

- Reading a 32-bit resource with value 1, a noValue resource, and a 32-bit resource with value 3:

  ResourceReadResult.data = [1, 3]

5.6.6 Writing a resource

The semantics of writing a resource are poke rather than bus write.

Side effects

Writing a resource should generally cause the side effect that a debugger user would expect when modifying the resource value. Side effects should be useful and intuitive, and should be kept to a minimum.

For most register resources, the side effect is the same as an architectural write.

For all resources where writes have side effects, these side effects, or the absence of them, must be documented in the resource description. For resources that expose multiple useful layers of side effects, multiple resources with intuitive but different names should be exposed. For example:

- A STATUS_CLEAR register whose only purpose is to clear another register when written should have this effect when written with resource_write().
- A STATUS register which clears itself when written to should not clear itself, but instead accept the value written to it. To clear the register, zero can be written to it.
- A TIMER register which you can modify to change the current timer value or which you can write to set a new reload value is best represented by three resources with different side effects:
  - TIMER for architectural writes.
  - TIMER_value to modify the timer value only.
  - TIMER_reload to modify the shadow timer reload register only.
Permissions and updating resources

Writing a resource should not be limited in any way. All bits that can architecturally change their value under certain conditions should be modifiable through `resource_write()`. However, writes that are architecturally forbidden and would lead to inconsistencies in the simulation state, should be ignored.

For example the following registers should be freely writable at all times:

- An EEPROM register containing a serial number which can normally only be programmed during a special reset procedure.
- A read-only flags register in which the flags can only be affected by executing instructions.
- A read-only cycle counter register, unless an update would cause inconsistent simulation state.
- An internal register that is architecturally inaccessible, for example an internal buffer or a shadow register.

Writes to read-only bits

Some or all bits of a resource might be read-only. Writes to these bits are ignored without error. If the whole resource is read-only, the `ResourceWriteResult.error` array should indicate this.

Write errors

`resource_write()` returns one `ResourceWriteResult` object. Errors that occurred while updating resources are returned in the `ResourceWriteResult.error` array. `resource_write()` never fails with an error when writing existing resources.

For examples of data and strings arguments, see 5.6.5 Reading a resource on page 5-69. If the values array is too long or too short for the specified resources, `resource_write()` returns `E_data_size_error`. In this case, the implementation might have updated any, or none of the specified resources.

5.6.7 ElfDwarf scheme for canonical register numbers

The ElfDwarf scheme is used for the `RegisterInfo.canonicalRn` field if the instance property `register.canonicalRnScheme` has the string value `ElfDwarf`.

The ElfDwarf scheme uses the DWARF register numbers defined for a specific architecture, combined with the value of the ELF header field, `e_machine`, which defines the architecture and is used as a namespace for the DWARF register numbers. This results in unique canonical register numbers across all architectures that ELF supports. The ElfDwarf scheme is defined for all architectures supported by ELF that define DWARF register numbers. See `/usr/include/elf.h` for a list of all possible values for `e_machine`.

Only a subset of the registers of an instance have an assigned DWARF register number. All other registers do not have a `RegisterInfo.canonicalRn` field and therefore cannot be discovered through the canonical register number. Instead, they can be discovered by inspecting `ResourceInfo.tags`, for instance `tags.isPc`, or `ResourceInfo.name`, which contains the architectural register name, if available.

The `RegisterInfo.canonicalRn` value is a 64-bit value with the following structure:

```
0x0000MMMM0000NNNN
```

Where:

- **Bits[15:0]**, **NNNN**
  - DWARF register number for the architecture that is defined in the ELF header field `e_machine`.

- **Bits[31:16]**, **0000**
  - Reserved. Instances must set these bits to zero.

- **Bits[47:32]**, **MMMM**
  - ELF `EM_*` constant for the architecture, as defined by the ELF header field `e_machine`. This is the namespace for the DWARF register number specified in bits[15:0].
Reserved. Instances must set these bits to zero.

Related references
5.6.8 ElfDwarf canonical register numbers on page 5-72

Related information
ELF Header
DWARF for the ARM Architecture
DWARF for the ARM 64-bit Architecture (AArch64)

5.6.8 ElfDwarf canonical register numbers

This table lists the canonical register numbers that Arm cores expose in ResourceInfo.registerInfo.canonicalRn, using the ElfDwarf scheme.

Note

- The C++ IrisSupportLib header file, iris/IrisElfDwarfArm.h contains symbolic constants for the canonical register numbers. They are listed in the Constant column.
- Only a small subset of the registers that Arm cores expose have canonical register numbers and DWARF register numbers assigned. Other registers can be discovered by name, for example PC, or by tag, for example ResourceInfo.registerInfo.tags.isPc.

<table>
<thead>
<tr>
<th>canonicalRn value</th>
<th>Register</th>
<th>Arch name</th>
<th>Arch number</th>
<th>Dwarf register</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2800000000+n</td>
<td>Rn (n=0-15)</td>
<td>EM_ARM</td>
<td>40</td>
<td>0+n</td>
<td>iris::ElfDwarf::ARM_R0+n</td>
</tr>
<tr>
<td>0x2800000080</td>
<td>SPSR</td>
<td>EM_ARM</td>
<td>40</td>
<td>128</td>
<td>iris::ElfDwarf::ARM_SPSR</td>
</tr>
<tr>
<td>0x2800000081</td>
<td>SPSR_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>129</td>
<td>iris::ElfDwarf::ARM_SPSR_fiq</td>
</tr>
<tr>
<td>0x2800000082</td>
<td>SPSR_irq</td>
<td>EM_ARM</td>
<td>40</td>
<td>130</td>
<td>iris::ElfDwarf::ARM_SPSR_irq</td>
</tr>
<tr>
<td>0x2800000083</td>
<td>SPSR_abt</td>
<td>EM_ARM</td>
<td>40</td>
<td>131</td>
<td>iris::ElfDwarf::ARM_SPSR_abt</td>
</tr>
<tr>
<td>0x2800000084</td>
<td>SPSR_und</td>
<td>EM_ARM</td>
<td>40</td>
<td>132</td>
<td>iris::ElfDwarf::ARM_SPSR_und</td>
</tr>
<tr>
<td>0x2800000085</td>
<td>SPSR_svc</td>
<td>EM_ARM</td>
<td>40</td>
<td>133</td>
<td>iris::ElfDwarf::ARM_SPSR_svc</td>
</tr>
<tr>
<td>0x2800000097</td>
<td>R8_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>151</td>
<td>iris::ElfDwarf::ARM_R8_fiq</td>
</tr>
<tr>
<td>0x2800000098</td>
<td>R9_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>152</td>
<td>iris::ElfDwarf::ARM_R9_fiq</td>
</tr>
<tr>
<td>0x2800000099</td>
<td>R10_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>153</td>
<td>iris::ElfDwarf::ARM_R10_fiq</td>
</tr>
<tr>
<td>0x280000009a</td>
<td>R11_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>154</td>
<td>iris::ElfDwarf::ARM_R11_fiq</td>
</tr>
<tr>
<td>0x280000009b</td>
<td>R12_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>155</td>
<td>iris::ElfDwarf::ARM_R12_fiq</td>
</tr>
<tr>
<td>0x280000009c</td>
<td>R13_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>156</td>
<td>iris::ElfDwarf::ARM_R13_fiq</td>
</tr>
<tr>
<td>0x280000009d</td>
<td>R14_fiq</td>
<td>EM_ARM</td>
<td>40</td>
<td>157</td>
<td>iris::ElfDwarf::ARM_R14_fiq</td>
</tr>
<tr>
<td>0x280000009e</td>
<td>R13_irq</td>
<td>EM_ARM</td>
<td>40</td>
<td>158</td>
<td>iris::ElfDwarf::ARM_R13_irq</td>
</tr>
</tbody>
</table>

---

a Canonical register number value in ResourceInfo.registerInfo.canonicalRn, according to the ElfDwarf scheme.
b Architectural register name.
c ELF EM_\* constant name for the architecture for which the register is defined.
d Numerical value of the EM_\* constant.
e DWARF register number defined for the architecture.
f Constant defined in iris/IrisElfDwarfArm.h for canonicalRn (uint64_t).
<table>
<thead>
<tr>
<th>canonicalRn value</th>
<th>Register</th>
<th>Arch name</th>
<th>Arch number</th>
<th>Dwarf register</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x280000009f</td>
<td>R14_irq</td>
<td>EM_ARM</td>
<td>40</td>
<td>159</td>
<td>iris::ElfDwarf::ARM_R14_irq</td>
</tr>
<tr>
<td>0x28000000a0</td>
<td>R13_abt</td>
<td>EM_ARM</td>
<td>40</td>
<td>160</td>
<td>iris::ElfDwarf::ARM_R13_abt</td>
</tr>
<tr>
<td>0x28000000a1</td>
<td>R14_abt</td>
<td>EM_ARM</td>
<td>40</td>
<td>161</td>
<td>iris::ElfDwarf::ARM_R14_abt</td>
</tr>
<tr>
<td>0x28000000a2</td>
<td>R13_und</td>
<td>EM_ARM</td>
<td>40</td>
<td>162</td>
<td>iris::ElfDwarf::ARM_R13_und</td>
</tr>
<tr>
<td>0x28000000a3</td>
<td>R14_und</td>
<td>EM_ARM</td>
<td>40</td>
<td>163</td>
<td>iris::ElfDwarf::ARM_R14_und</td>
</tr>
<tr>
<td>0x28000000a4</td>
<td>R13_svc</td>
<td>EM_ARM</td>
<td>40</td>
<td>164</td>
<td>iris::ElfDwarf::ARM_R13_svc</td>
</tr>
<tr>
<td>0x28000000a5</td>
<td>R14_svc</td>
<td>EM_ARM</td>
<td>40</td>
<td>165</td>
<td>iris::ElfDwarf::ARM_R14_svc</td>
</tr>
<tr>
<td>0x28000000100+n</td>
<td>Dn (n=0-31)</td>
<td>EM_ARM</td>
<td>40</td>
<td>256+n</td>
<td>iris::ElfDwarf::ARM_D0+n</td>
</tr>
<tr>
<td>0xb700000000+n</td>
<td>Xn (n=0-30)</td>
<td>EM_AARCH64</td>
<td>183</td>
<td>0+n</td>
<td>iris::ElfDwarf::AARCH64_X0+n</td>
</tr>
<tr>
<td>0xb70000001f</td>
<td>SP</td>
<td>EM_AARCH64</td>
<td>183</td>
<td>31</td>
<td>iris::ElfDwarf::AARCH64_SP</td>
</tr>
<tr>
<td>0xb70000021</td>
<td>ELR</td>
<td>EM_AARCH64</td>
<td>183</td>
<td>33</td>
<td>iris::ElfDwarf::AARCH64_ELRR</td>
</tr>
<tr>
<td>0xb70000040+n</td>
<td>Vn (n=0-31)</td>
<td>EM_AARCH64</td>
<td>183</td>
<td>64+n</td>
<td>iris::ElfDwarf::AARCH64_V0+n</td>
</tr>
</tbody>
</table>

**Related concepts**

5.6.7 ElfDwarf scheme for canonical register numbers on page 5-71

**Related information**

ELF Header

DWARF for the ARM Architecture

DWARF for the ARM 64-bit Architecture (AArch64)

### 5.6.9 Comparison of resource types

It is generally clear from the context whether a piece of state that an instance exposes is an init-time parameter, a run-time parameter, or a register. If it is unclear, see the following table.

In this table, if the criterion is true, the value can be modeled as shown, reading from left to right:

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Init-time parameter</th>
<th>Run-time parameter</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is configurable at initialization time.</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Is modifiable at runtime.</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Is modifiable at runtime and generally does not change at runtime</td>
<td>no</td>
<td>yes</td>
<td>usually no</td>
</tr>
<tr>
<td>except when the user changes it.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can change spontaneously, that is, without a parameter write, at runtime.</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Is an architectural register.</td>
<td>no</td>
<td>no</td>
<td>potentially</td>
</tr>
</tbody>
</table>

---

a  Canonical register number value in ResourceInfo.registerInfo.canonicalRn, according to the ElfDwarf scheme.

b  Architectural register name.

c  ELF EM_* constant name for the architecture for which the register is defined.

d  Numerical value of the EM_* constant.

e  DWARF register number defined for the architecture.

f  Constant defined in iris/ElfDwarfArm.h for canonicalRn (uint64_t).
Table 5-3 Comparison of resource types (continued)

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Init-time parameter</th>
<th>Run-time parameter</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Has a reset value.</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Is reset during simulation reset.</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Corresponds to a design-time parameter of the hardware.</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Corresponds to a hardware register.</td>
<td>no</td>
<td>no</td>
<td>potentially</td>
</tr>
<tr>
<td>Can be modified by program code.</td>
<td>no</td>
<td>no</td>
<td>usually yes</td>
</tr>
<tr>
<td>Is artificial, not present in the hardware.</td>
<td>yes</td>
<td>yes</td>
<td>usually no</td>
</tr>
<tr>
<td>Can have fields.</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Is organized into groups.</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

5.6.10 Exposure of parameters

Parameters are exposed through the following functions:

resource_getList()

Returns all initialization-time parameters, run-time parameters, registers, and generic resources.

resource_read()

Returns the current value of initialization-time parameters, run-time parameters, registers, and generic resources.

resource_write()

Sets run-time parameters, registers, and generic resources. Returns E_writing_init_time_parameter in ResourceWriteResult.error when trying to write an initialization-time parameter.

simulation_getInstantiationParameterInfo()

Returns all initialization-time parameters and all run-time parameters. Does not return non-parameter resources.

simulation_setInstantiationParameterValues()

Sets initialization-time parameters and run-time parameters. Does not set non-parameter resources.

Initialization-time parameters are exposed by:

- resource_getList()
- resource_read()
- simulation_getInstantiationParameterInfo()
- simulation_setInstantiationParameterValues()
- resource_write() causes an error.

Run-time parameters are exposed by:

- resource_getList()
- resource_read()
- resource_write()
- simulation_getInstantiationParameterInfo()
- simulation_setInstantiationParameterValues()

Registers are exposed by:

- resource_getList()
- resource_read()
- resource_write()
5.6.11 resource_getList()

Retrieves the static aspects of a resource. Debuggers usually call this function only once, after connecting to the target. If neither group nor rscId arguments are specified, all resources are returned.

Arguments

- **group**
  - Type: String
  - Optional. Return information just for resources that are part of this resource group. If no such group is known, E_unknown_resource_group is returned. If a valid rscId is also specified, in addition to a group, only a single matching resource or an empty array (and no error) is returned.

- **instId**
  - Type: NumberU64
  - Opaque number uniquely identifying the target instance.

- **rscId**
  - Type: NumberU64
  - Optional. Return information just for the resource with rscId. The return value is an array with a single element. If rscId is not known, E_unknown_resource_id is returned. If group is also specified, the resource is just searched for in the group. If a valid rscId is not part of the specified group, an empty array and no error is returned.

Return value

- ResourceInfo[]
  - Zero or more ResourceInfo on page 5-80 Objects.

Errors

- E_unknown_instance_id.
- E_unknown_resource_id.
- E_unknown_resource_group.

5.6.12 resource_getListOfResourceGroups()

Gets the meta information of all resource groups. Each resource group has a name, a description, and a list of ids of the resources in the group.

Arguments

- **instId**
  - Type: NumberU64
  - Opaque number uniquely identifying the target instance.

Return value

- ResourceGroupInfo[]
  - Zero or more ResourceGroupInfo on page 5-79 Objects.
Errors
- E_unknown_instance_id.

5.6.13 resource_read()

Reads the values of a set of resources. The semantics are peek rather than architectural read.

Arguments
instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

rscIds
Type: NumberU64[]
List of opaque resource ids uniquely identifying the resources within the target instance to be read. An empty array is valid and results in an empty result array. An rscId can occur multiple times, in which case the same resource is read multiple times.

Return value
ResourceReadResult
Resource data, undefined bits, and errors. See ResourceReadResult on page 5-84.

Errors
- E_unknown_instance_id.
- E_unknown_resource_id.

5.6.14 resource_write()

Writes values to a set of resources. The semantics are poke rather than architectural write.

Arguments
data
Type: NumberU64[]
List of numeric resource values to be written, in the order they are specified in the rscIds argument. See ResourceReadResult.data on page 5-84 for the encoding. If this array does not match the number of resources being written, E_data_size_error is returned.

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

rscIds
Type: NumberU64[]
List of opaque resource ids uniquely identifying the resources within the target instance to be written. An empty array is valid. The same rscId can occur more than once, in which case the same resource is written multiple times.

strings
Type: String[]
Optional. List of string resource values. Missing if no string resources are written. See ResourceReadResult.strings on page 5-84 for the encoding. If this array does not match the number of resources being written, E_data_size_error is returned.
Return value

ResourceWriteResult

Write errors. See ResourceWriteResult on page 5-87.

Errors

• E_unknown_instance_id.
• E_unknown_resource_id.
• E_data_size_error.

5.6.15 ParameterInfo

ParameterInfo members:

defaultValue

Type: NumberU64[]

Optional. Default value of the numeric parameter. Parameters for which no specific value was
specified at init-time have this value. If this is not specified, a default value of 0 is used. This is
only present for numeric parameters. Signed numeric values have the (unsigned) bit pattern in
the default value. numericFp resources have floating-point bit patterns in the default value.

Numeric values are encoded right-aligned into the NumberU64 elements and little-endian in the
array (the same encoding when reading or writing resources). Instances must guarantee that the
default value is consistent with the min and max values, but clients must not rely on this
guarantee.

defaultString

Type: String

Optional. Default value of a string parameter. Parameters for which no specific value was
specified at init-time have this value. If this is not specified, by default an empty string is used.

initOnly

Type: Boolean

Optional. Specifies when a parameter value can be set or written:

True

Init-time parameter

False

Run-time parameter. Default is false (run-time parameter).

Init-time parameter: The parameter can only ever be set before instantiation. The parameter is
only settable through the function simulation_setInstantiationParameterValues() on page 5-192. It is not settable at run-time through the function resource_write() on page 5-76, which is only available after instantiation. The parameter behaves like a read-only
resource after instantiation. After instantiation, it can be read through resource_read() on page 5-76 like any other resource. The value of init-time parameters must be constant at run-
time. Typical examples of init-time parameters are cache sizes or number of sub-units.

Run-time parameter: The parameter can be set at init-time and it can also be set at run-time. The
parameter mostly behaves like a generic resource. As a guideline, the value of a run-time
parameter must not change spontaneously at run-time other than through explicit
resource_write() on page 5-76 calls or equivalent explicit mechanisms, for example re-
reading a config file. If it does change spontaneously at run-time, it is likely to be better modeled as a non-parameter resource.

Note

Even if a parameter is only modifiable at run-time under very restricted circumstances, it must be a run-time parameter. Run-time parameters are not guaranteed to be modifiable at run-time, but init-time parameters are guaranteed to be constant at run-time.

Note

All parameters, regardless of the value of `initOnly`:

- Can always be set at init-time through `simulation_setInstantiationParameterValues()` on page 5-192.
- Can always be read at run-time through `resource_read()` on page 5-76.
- Are exposed through `simulation_getInstantiationParameterInfo()` on page 5-191.

---

max

Type: `NumberU64[]`

Optional. Maximum numeric value accepted (inclusive). Only used for numeric types.

min

Type: `NumberU64[]`

Optional. Minimum numeric value accepted (inclusive). Only used for numeric types.

5.6.16 RegisterInfo

RegisterInfo members:

addressOffset

Type: `NumberU64`

Optional. Address offset of a memory-mapped register, relative to the physical base address of the instance, for example a peripheral. This is informational only. A lot of factors affect the memory map of a system. Clients can safely ignore the value of `addressOffset` and usually just treat it as part of the register description.

canonicalRn

Type: `NumberU64`

Optional. Canonical register number. This number adheres to the scheme described by the register. `canonicalRnScheme` property returned by `instance_getProperties()`.

If register. `canonicalRnScheme` is `ElfDwarf` then bits[15:0] specify the DWARF register number for the architecture that is specified by bits[47:32]. The architectures are as defined in the ELF header field `e_machine`. See section "ElfDwarf scheme for canonical register numbers" for details.
lsbOffset

Type: NumberU64

Optional. LSB offset of this register in its parent register specified by parentRscId. Together with bitWidth this describes the bit range covered by this register field in its parent register. If lsbOffset is present, parentRscId must also be set. For logical child registers, on the other hand, it is valid to have just parentRscId set and lsbOffset not present. This indicates that this register is a logical child register of its parent, not covering a simple consecutive bit range. This can be used to cleanly expose child registers that are distributed across several bit ranges in the parent registers, reordering and splitting bits arbitrarily. It can also be used to expose logical parts of a parent register that do not appear in the bits of the parent register at all.

The special value '2^64-1' can be used synonymously to a missing value. This is for cases where not specifying lsbOffset is not possible, for example arrays of the same object signature or C++.

Clients can safely ignore the value of lsbOffset and usually just treat it as part of the register description. Clients do not need to interpret lsbOffset in any way to read and/or write register values. The target instance takes care of putting the right bits into the right location.

resetData

Type: NumberU64[]

Optional. Reset value for a numeric register. This is the value to which the resource is set at hardware reset. This is informational only. It is up to the instance implementation to implement a correct reset behavior. Registers that do not have a defined reset value must not have a resetData field. The value is encoded right-aligned inside each NumberU64 and little-endian in the array, the same format that is used for resource values used by 'resource_read()' and 'resource_write()'. Clients can safely ignore the value of resetData and usually just treat it as part of the register description.

resetString

Type: String

Optional. Reset value for string resource. See resetData.

writeMask

Type: NumberU64[]

Optional. Write mask. Bits set in this mask can be written. Other bits cannot be modified. This is informational only. It is up to the instance implementation to enforce the write mask and is not guaranteed by the interface. Clients must not enforce the write mask when writing the resource. The value is encoded right-aligned inside each NumberU64 and little-endian in the array, the same format that is used for resource values used by 'resource_read()' and 'resource_write()'. Clients can safely ignore the value of writeMask and usually just treat it as part of the register description.

5.6.17 ResourceGroupInfo

ResourceGroupInfo members:
cname
Type: String
Name of the resource group in the context of expressions and in scripts. This is a valid C identifier that is unique within this instance and that also does not overlap with resource cname fields. Both name and cname are primary keys to identify groups. Instance implementations derive the cname from name according to the rules specified for ResourceInfo.cname on page 5-80 if it is not explicitly specified when defining a resource. Clients must handle violations of these rules gracefully, converting cname to a C identifier according to the rules specified in ResourceInfo.cname on page 5-80 (which leaves all valid C identifiers untouched). In case of conflicts with other groups cnames only the first group within the array returned by 'resource_getListofResourceGroups()' with a specific cname must be accessible. In case of a conflict with resource cnames the group cname must always take precedence.

description
Type: String
Optional. Description of the resource group. Can contain linefeeds.

name
Type: String
Short string identifying the resource group. This is unique within this instance and must not overlap with resource name fields. Both name and cname are primary keys to identify groups.

resourceList
Type: NumberU64[]
List of resource ids of the resources that belong to this group. Must not be empty. Each resource must be in at least one group, and can be in more than one group. The array defines the order of the resources in each group that clients must use when displaying the resources of a group.

5.6.18 ResourceInfo
ResourceInfo members:

bitWidth
Type: NumberU64
Size of the resource in bits. Must be 0 for type="string" and type="noValue" resources. Must be > 0 for all other types of resources.
cname

Type: String

Name of the resource in the context of expressions and in scripts. This is a valid C identifier. For top-level resources this must be unique within each resource group this resource belongs to. For child resources this must be unique within the parent resource. Instance implementations derive the cname from name according to the following rules if no explicit cname is specified when defining a resource:

- All non-C identifier chars are replaced with underscores.
- If the first char is a digit, an underscore is prepended.

Clients must handle violations of these rules gracefully, by converting cname to a C identifier according to the rules specified above (which leaves all valid C identifiers untouched). In case of conflicts with other resource cnames only the first resource within the array returned by 'resource_getList()' with a specific cname must be accessible.

Child registers do not contain the cnames of their parents in their cname. Clients that need a hierarchical C identifier for a child register must prepend the cnames of all parent registers, separated with an underscore (for instance 'FLAGS_X'), to create a hierarchical cname. See also name.

description

Type: String

Optional. Description of the semantics of the resource. Might contain linefeeds.

enums

Type: EnumElementInfo[

Optional. Array of EnumElementInfo on page 5-186 objects which describe symbols for numeric resource values. Debuggers can display these symbols (and potentially their description) in addition to the numeric value.

format

Type: String

Optional. Iris-text-format. This allows a client to format the value of this resource and the values of other resources in this instance (regardless of hierarchy) in a specific way. The value of this resource is referred to by the value variable in the Iris-text-format string. The values of other resources in this instance are referred to by their cname, optionally prefixed by their group cname and a dot (resource name without group name has priority). Clients must allow the user to select whether the format must be used (if present at all) or whether the plain numeric value must be displayed. Clients must prefer the format if present. Resources that only pull together information from other resources and do not have any value on their own can set type="noValue" to make this clear. This can for example be used to format bitfields in a descriptive way. Cache line tag: "Addr=%{value[31:12]}000 %{value[11]:clean|dirty}". Conditional formatting is also possible.
name

Type: String

Display name of the resource. For registers, this must be the architectural register name if available. For top-level resources this must be unique within each resource group this resource belongs to. For child resources this must be unique within the parent resource. Clients must handle non-unique register names gracefully. They must always use the first resource in the array returned by `resource_getList()` on page 5-75 in case of a conflict.

The resource group name is not part of this resource name.

Child resources (which have a `parentRscId` field) specify only their own name in the `name` and `cname` fields, not including the names of their parents. Clients that need a hierarchical name for a child register must prepend the names of all parent registers, separated with a dot, to get a hierarchical name (for instance 'FLAGS.X').

See also `cname` which is used in expressions and scripts.

parameterInfo

Type: ParameterInfo

Optional. Iff this is present, this resource represents a parameter. Parameters generally behave like normal resources and have some additional semantics attached to them, like being settable at init-time. This `ParameterInfo` on page 5-77 object contains parameter specific meta information. It also acts as the `isParameter` switch.

parentRscId

Type: NumberU64

Optional. If this is present, this is a child resource (for instance a field in a register) and this field contains the resource id of its parent resource. The `name` and `cname` fields of child resources only contain the names of the child resources, not of the parent resources.

Child resources might, in turn, have child resources. The nesting depth is not limited, but instances are encouraged to expose only shallow hierarchies (usually only one level). Instances must not expose loops in the parent graph. Clients must handle the situation where an instance exposes a loop in the parent graph gracefully, for instance by ignoring any parent pointer that points to a child. Clients must display child resources in the order they appear in the array returned by `resource_getList()` on page 5-75 underneath a parent.

If `parentRscId` is missing this means that this is a top-level resource (which might or might not have children). All children of a specific parent resource must be in the same group as the parent resource.

registerInfo

Type: RegisterInfo

Optional. Iff this is present, this resource represents a register. Registers typically correspond to architectural or device registers in the component and have some additional metadata found in the `registerInfo` object.

rscId

Type: NumberU64

Opaque resource id uniquely identifying the resource within the target instance. Used to read/write the resource. This is the primary key to identify a resource. The value $2^{64}-1$ is defined to be an invalid rscId.
**rwMode**

Type: String

Optional. Either "r", "w", or "rw" for read-only, write-only, or read-write (default). This is a hint for the debugger or user about which accesses on this resource are supported architecturally. However, it is always allowed to issue `resource_read()` on page 5-76 and `resource_write()` on page 5-76 calls on all resources, regardless of their rwMode. This must not cause an error in the `resource_read()` on page 5-76 or `resource_write()` on page 5-76 return value. This might or might not cause errors in the 'ResourceReadResult.error' array (E_error_reading_write_only_resource, E_error_writing_read_only_resource, E_error_reading_resource, E_error_writing_resource).

Ultimately, the instance implementation decides what happens for a read or a write. Writes to read-only resources can be silently ignored or can return E_error_writing_read_only_resource and reads of write-only resources must return a useful value, for example the value of an internal latch, or can return E_error_reading_write_only_resource.

**subRscId**

Type: NumberU64

Optional. Additional resource id according to an instance-specific scheme defined by the instance. This is not required to be unique within an instance, nor is it required to be present for every resource of an instance. This is usually just used internally in an instance to identify registers in register access functions and is rarely useful for clients.

--- Note ---

For register resources, address offsets and canonical register numbers, which both are also suitable to identify registers, must go into the RegisterInfo addressOffset or canonicalRn fields, respectively. subRscId uses a scheme that is private to an instance.

---

**tags**

Type: ResourceTags

Optional. Object containing extra meta information about a resource, for example whether a register is the PC or stack pointer, see the ResourceTags on page 5-85 object. If this is missing, this has the same semantics as specifying an empty ResourceTags on page 5-85 object. This means the resource is not tagged with anything, which is usually the case for most resources of an instance.
**type**

Type: String

Optional. Either "numeric", "numericSigned", "numericFp", "string" or "noValue". Default is "numeric".

"numeric": The resource contains a bit-pattern of "bitWidth" bits. Clients must display this as hex by default and allow users to explicitly switch to other formats, for example decimal or floating point. This is the usual type for most resources. All other resource types are exotic.

"numericSigned": Exotic: Same as "numeric" but in addition this gives a hint to the client that this resource always contains a signed integer. This is usually used for parameters that represent a signed integer value.

"numericFp": Exotic: Same as "numeric" but in addition this gives a hint to the client that this resource always contains an IEEE 754 floating-point value. This is usually only useful for pure floating-point resources. Clients must display this as a floating-point number by default and allow users to switch explicitly to hex and other formats. This must only be used for bitWidth 32 (float) and bitWidth 64 (double). Clients must treat other bitWidths as "numeric".

"string": Exotic: The resource contains an ASCII string. Targets can use this type as a fallback to display arbitrary information in the debugger, or as parameters that have a string value, like filenames. bitWidth must be 0 for string resources. This must not be used to expose arbitrary length binary data. To expose arbitrary length binary data consider using a memory space or a table if the data is structured.

"noValue": Exotic: No numeric or string value is associated with this resource. If this resource has a format then clients must display this format, usually by pulling together numeric information from other resources. bitWidth must be 0. Reading a noValue resource always delivers zero NumberU64 units in ResourceReadResult.data on page 5-84 and ResourceReadResult.undefinedBits on page 5-84. Writing a noValue resource is always silently ignored. Clients might or might not allow editing of noValue resources that have a format string. Resource breakpoints can be supported for noValue resources.

**Related references**

5.5 Iris-text-format on page 5-64

5.6.19 ResourceReadResult

ResourceReadResult members:

data

Type: NumberU64[]

List of numeric resource values read, in the order they were specified in the rscIds argument of the resource_read() on page 5-76 function. The encoding of the value depends on the type:

**numeric, 0<bitWidth<=64**

NumberU64, value right-aligned in the lowest bits, zero-extended to 64 bits.

**numeric, bitWidth>64**

Sequence of N*NumberU64 in little-endian, last element right-aligned in the lowest bits, zero-extended to N*64 bits, N = floor((bitWidth+63)/64).

**numericSigned**

Same as numeric except that the value is sign-extended instead of zero-extended.

**numericFp**

Same as numeric.

**string**

Not present in this array. Occupies zero elements in this array.
noValue

Not present in this array. Occupies zero elements in this array. All numeric resources read have a value in this array (potentially a dummy value), regardless of whether the read operation failed or succeeded (regardless of whether there is an entry for a resource in the error array). The layout of the data array can be interpreted in the same way, just based on the layout of the rscIds array in the `resource_read()` on page 5-76, regardless of the contents of errors. The length of this array depends on the number and width of all numeric resources read.

error

Type: `NumberU64[]`

Optional. List of resource ids and E_* error codes for existing resources that could not be read. Each entry in the list occupies two array elements: rscId and errorCode. This only returns errors that happen because the callee is unable to provide the resource value of an existing resource, for example architectural errors. This does not return errors caused by the caller doing something wrong, for example E_unknown_resource_id. These errors are returned by `resource_read()` on page 5-76 instead. The errors that are most likely to occur in this list are:

E_approximation

The resource value could be read but is only an approximation in some sense, for example because the instance is not in a debuggable-state.

E_value_not_available

The resource value is currently not available and cannot even be approximated, for example because the instance is not in a debuggable-state.

""E_error_reading_write_only_resource", "E_error_reading_resource" (catch all)"

(These errors must be shown as information to the user (similar to undefined bits), not as a user or implementation error. Note)

The error codes E_unknown_instance_id and E_unknown_resource_id are never returned in this array, but are instead returned by the `resource_read()` on page 5-76 function. This array is either missing, in case of no such error, or non-empty, in case of errors.

strings

Type: `String[]`

Optional. List of string resource values read, in the order they were specified in the rscIds argument of the `resource_read()` on page 5-76 function. If no string resources were read, this array is missing. All string resources have a string value in this array, regardless of whether the read operation failed or not. See "data".

undefinedBits

Type: `NumberU64[]`

Optional. List of resource masks that indicate undefined bits. This array has the same layout and length as data, even if only a subset of the read resources support undefined bits. A 1-bit means that the value is undefined, in which case the corresponding bit in data is 0 and must be ignored by clients. Undefined bits are only reported by instances that support it and only for resources that support it. A missing array is equivalent to an array containing all zeros (all bits defined).

5.6.20 ResourceTags

ResourceTags members:
isPc
Type: Boolean
Optional. If present and True, this is the register that represents the address that is executed next of a CPU or code-executing component. At most one register must have this set to True. Debuggers use this to find out which instruction or line is to be executed next. The isPc and the isPcSpaceId resources together describe the next execution location.

isSp
Type: Boolean
Optional. If present and True, this is the stack pointer of a CPU. At most one register must have this set to True.

isLr
Type: Boolean
Optional. If present and True, this is the link register of a CPU. At most one register must have this set to True.

isFramePointer
Type: Boolean
Optional. If present and True, this is the frame pointer register of a CPU. At most one register must have this set to True.

isPcSpaceId
Type: Boolean
Optional. If present and True, this resource contains the memory space id for the PC and for the link register. At most one resource must have this set to True. The isPc and the isPcSpaceId resources together describe the next execution location.

isSpSpaceId
Type: Boolean
Optional. If present and True, this resource contains the memory space id for the stack pointer and frame pointer. At most one resource must have this set to True.

isInstructionCounter
Type: Boolean
Optional. If present and True, this resource is the instruction counter. The instruction counter counts executed instructions linearly from 0. This is not the PC.
isArchitectural

Type: Boolean

Optional. If present and True, this resource is an architectural register. An architectural register is a register that is mentioned in the architecture reference manual for an instance. Architectural registers must use their architectural name.

Examples of architectural registers are the general purpose registers, the PC, and the flags register of a core. Examples of non-architectural register resources are artificial resources of simulation components that enable or disable certain features, count simulation events, or allow access to internal state, and registers that represent architectural information in a (potentially more convenient) non-architectural format. There is however no hard line between architectural and non-architectural registers, for example for internal shadow registers present in hardware and mentioned in the technical reference manual if an architecture reference manual is missing.

This flag is informational and clients can use it to extract the subset of architectural registers of an instance. It must be treated like a part of the description of a resource.

5.6.21 ResourceWriteResult

ResourceWriteResult members:

error

Type: NumberU64[]

Optional. List of resource ids and E_* error codes for existing resources that could not be written. This only returns errors that happen because the callee is unable to write the resource value of an existing resource, for example architectural errors. This does not return errors caused by the caller doing something wrong, for example E_unknown_resource_id. These errors are returned by resource_write() on page 5-76 instead. Each entry in the list occupies two array elements: rscId and errorCode. The errors that are most likely to occur in this list are E_error_writing_read_only_resource, E_writing_init_time_parameter (when trying to write an init-time parameter), and E_error_writing_resource (catch-all error if reading failed). These errors must be shown as information to the user (similar to undefined bits), not as a user or implementation error.

——— Note ————

The error codes E_unknown_instance_id and E_unknown_resource_id are never returned in this array, but are instead returned by the resource_write() on page 5-76 function. This array is either missing, in case of no such error, or non-empty, in case of errors.
5.7 Memory API

The memory interface allows you to access data in the memory spaces that an instance exposes. All memory spaces are assumed to be byte-addressable, in the sense that each address refers to a byte location.

Clients first query a list of available memory spaces and their meta information by calling `memory_getMemorySpaces()`. Each memory space is identified by a memory space id. Memory is read or written by using the `spaceId` and the `memory_read()` or `memory_write()` function. The other memory functions provide less common functionality, for example address translations and retrieving sideband information.

$IRIS_HOME/Examples/Client/Memory/ contains an example client application that demonstrates how to use this API.

This section contains the following subsections:
- 5.7.1 Memory accesses on page 5-88.
- 5.7.2 Errors on page 5-89.
- 5.7.3 Endianness on page 5-89.
- 5.7.4 Memory spaces on page 5-89.
- 5.7.5 Side effects on page 5-89.
- 5.7.6 Canonical memory space number scheme on page 5-89.
- 5.7.7 Memory access attributes on page 5-91.
- 5.7.8 Reading and writing memory on page 5-92.
- 5.7.9 Reading and writing through caches and buffers on page 5-93.
- 5.7.10 Address translation on page 5-93.
- 5.7.11 Memory sideband information on page 5-93.
- 5.7.12 memory_getMemorySpaces() on page 5-94.
- 5.7.13 memory_getSidebandInfo() on page 5-94.
- 5.7.14 memory_getUsefulAddressTranslations() on page 5-95.
- 5.7.15 memory_read() on page 5-96.
- 5.7.16 memory_translateAddress() on page 5-97.
- 5.7.17 memory_write() on page 5-97.
- 5.7.18 MemoryAddressTranslationResult on page 5-99.
- 5.7.19 MemoryReadResult on page 5-99.
- 5.7.20 MemorySpaceInfo on page 5-100.
- 5.7.21 MemorySupportedAddressTranslationResult on page 5-102.
- 5.7.22 MemoryWriteResult on page 5-102.

5.7.1 Memory accesses

The interface supports access widths that are a power of two bytes, most commonly one, two, four, or eight bytes, as specified in the `byteWidth` argument to `memory_read()` and `memory_write()`.

Instances are not required to support all access widths for all addresses. They can either return an error if elements could not be read or written or they can return zero for reads, and ignore writes.

All accesses must be naturally aligned, or `E_unaligned_access` is returned.

Memory locations are read from lower addresses to higher addresses. Accessing memory does not stop on read or write errors.

If the `count` argument to `memory_read()` or `memory_write()` is greater than one, instances can convert debug accesses covering multiple elements into burst accesses, if the bus supports it. Buses must break debug burst accesses into individual accesses transparently, for example when passing accesses to peripheral buses that do not support bursts. To reduce the function call overhead, clients should generally make the access count as large as possible.
5.7.2 Errors

Reading and writing memory can fail for various reasons, including data aborts, translation errors, unsupported byteWidth values, or reading past the end of the memory space.

Such errors do not cause the memory_read() or memory_write() function to return an error, but instead are reported in the error member of the MemoryReadResult or MemoryWriteResult return value.

Reads and writes specify a start address, an access width for each element (byteWidth), and a number of elements (count). The start address must be within the range supported by the memory space, in other words between minAddr and maxAddr of the memory space, or E_address_out_of_range is returned. However, the end address can be beyond the end of the memory space. In other words, address + (byteWidth*count)-1 is not required to be within minAddr to maxAddr. Reads and writes must return an error in the error member of the result for all elements that exceed the memory space address range.

Target instances that do not expose any memory must return E_function_not_supported_by_instance for the memory_*() functions.

5.7.3 Endianness

The target instance is responsible for using the endianness that is specified in the memory space when writing values to memory.

8-bit, 16-bit, and 32-bit numbers are packed into NumberU64 values with the lowest address starting at bit[0], in other words, little-endian, regardless of the endianness of the memory space. Values that are greater than or equal to 128 bits are packed into a sequence of NumberU64 with the lowest bits first, little-endian, regardless of the endianness of the memory space.

5.7.4 Memory spaces

All memory spaces are considered to be orthogonal to each other.

minAddr and maxAddr do not indicate the start and end of memory blocks inside a memory space, but rather the smallest and largest addresses supported by a memory space. This interface has no representation for memory blocks inside a memory space. Memory spaces usually start at address zero.

5.7.5 Side effects

The side effects of reading and writing memory are as follows:

• If possible, component creators should ensure that memory accesses are side-effect free. Side-effect free reads are a prerequisite for non-intrusive debug, although they might not be possible due to bugs in target instance implementations or bridges into environments that do not support them.

• Writing memory should be as free of side effects as possible. In other words, it should cause just enough side effects to keep the target and system state consistent. For memory-mapped registers, the side effect should be the same as calling resource_write().

• Reading from cached memory must not allocate into the cache or change the cache tag in any way. It must follow the cache hierarchy until a hit is found and return the data.

• Writing to cached memory must update all cache lines that hold the written memory location. It must write through to all cache levels including main memory, regardless of the allocation strategy of the cache. Write accesses must never allocate into the cache or change the cache tag in any way. Writes never set dirty bits of caches. Writes cannot cause a cache inconsistency, but they can remove cache inconsistencies.

5.7.6 Canonical memory space number scheme

All Arm components implement the canonical memory space number scheme arm.com/memoryspaces. It allows debuggers to programmatically select a specific translation regime. The semantics of some memory spaces depend on the Arm architecture version and even on the configuration of EL3.

Related references
5.6.14 resource_write() on page 5-76
The following ids are defined for the `canonicalMsn` member of `MemorySpaceInfo`:

### Table 5-4 Canonical memory space number scheme arm.com/memoryspaces

<table>
<thead>
<tr>
<th><code>canonicalMsn</code> (NumberU64)</th>
<th>Architecture and configuration</th>
<th>Name</th>
<th>Semantics and translation regime</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>Armv8 EL3=AArch64</td>
<td>Secure Monitor</td>
<td>Virtual memory as seen by code running at EL3. This is always secure. This is virtual memory as configured by TCR_EL3.</td>
</tr>
<tr>
<td></td>
<td>Armv7, Armv8, EL3=AArch32</td>
<td>Secure Monitor</td>
<td>Virtual memory as seen by code running at PL0 or PL1 on the secure side. This is always secure. This is virtual memory as configured by TTBCR(secure).</td>
</tr>
<tr>
<td></td>
<td>Armv6, Armv7</td>
<td>Secure</td>
<td>Virtual memory as configured by TTBCR(secure).</td>
</tr>
<tr>
<td>0x1001</td>
<td>Armv8 EL3=AArch64</td>
<td>Guest</td>
<td>Virtual memory as seen by code running at EL0 or EL1. This can be secure or non-secure. This is virtual memory as configured by TCR_EL1/TTBCR(non-secure).</td>
</tr>
<tr>
<td></td>
<td>Armv7, Armv8, EL3=AArch32</td>
<td>Guest</td>
<td>Virtual memory as seen by code running at PL0 or PL1 on the non-secure side. This is always non-secure. This is virtual memory as configured by TTBCR(non-secure).</td>
</tr>
<tr>
<td></td>
<td>Armv6, Armv7</td>
<td>Normal</td>
<td>This is virtual memory as configured by TTBCR(non-secure).</td>
</tr>
<tr>
<td>0x1002</td>
<td>Armv7, Armv8</td>
<td>NS Hyp</td>
<td>Virtual memory as seen by code running at EL2/PL2, for AArch64/AArch32. This is always non-secure. This is memory as configured by TCR_EL2/HTCR, for AArch64/AArch32.</td>
</tr>
<tr>
<td>0x1003</td>
<td>Armv5, Armv6, Armv7</td>
<td>Memory</td>
<td>Virtual memory. Cores and other components that do not have TrustZone®.</td>
</tr>
<tr>
<td>0x1004</td>
<td>Armv7, Armv8</td>
<td>Hyp App</td>
<td>Virtual memory as seen from EL0 (32 or 64) running under a hypervisor with HCR.TGE=1. This has stage1 implicitly disabled but is still translated by stage2.</td>
</tr>
<tr>
<td>0x1005</td>
<td>Armv8.1</td>
<td>Host</td>
<td>Virtual memory as seen from EL0 (32 or 64) and EL2 (64) with HCR.E2H=1 and HCR.TGE=1. This has stage1 controlled by TCR_EL2 and implicitly disables stage2.</td>
</tr>
<tr>
<td>0x10ff</td>
<td>All</td>
<td>Current</td>
<td>Virtual memory view of the current exception level, protection level, or mode. The translation regime used follows the current state of the CPU.</td>
</tr>
<tr>
<td>0x1100</td>
<td>Armv7, Armv8</td>
<td>IPA</td>
<td>Intermediate physical memory view. Non-secure.</td>
</tr>
<tr>
<td>0x1200</td>
<td>Armv6, Armv7, Armv8</td>
<td>Physical Memory (Secure)</td>
<td>Physical memory, secure world.</td>
</tr>
<tr>
<td>0x1201</td>
<td>Armv6, Armv7, Armv8</td>
<td>Physical Memory (Non Secure)</td>
<td>Physical memory, non-secure world.</td>
</tr>
<tr>
<td>0x1202</td>
<td>Armv5, Armv6, Armv7</td>
<td>Physical Memory</td>
<td>Physical memory. Cores and components that do not have TrustZone.</td>
</tr>
</tbody>
</table>
Entries in the Name column are for information only and are not binding. They reflect the names that are used by existing models. The purpose of the canonicalMsn is for clients to use it to find a memory space with specific semantics.

Armv8 instances support the AArch64 and AArch32 modes at runtime for the memory spaces with canonicalMsn = 0x1000, 0x1001, and 0x1002. These memory spaces should have the static properties of AArch64, with 64-bit wide virtual addresses.

### 5.7.7 Memory access attributes

The set of memory access attributes that are available depends on the target instance and the memory space within it. The supported attributes and their semantics are listed in the target instance documentation and are also provided by the attrib and attribDefaults members of MemorySpaceInfo.

However, there are typical classes of instances and memory spaces that expose the same set of attributes. The following tables list all possible memory attributes as a guideline for instance implementations.

There are no attributes for generic storage components like RAM, ROM, flash, and other backing storage.

The translation regimes that are implemented by the CPU should be exposed as memory spaces. All virtually-addressed regimes should be exposed, if applicable. In addition, a physical view of the memory should be exposed as a memory space, if applicable.

See 5.7.6 Canonical memory space number scheme on page 5-89 for a list of canonical memory spaces. Each memory space has a different set of default values for these attributes, which often define the semantics of the memory space.

#### Table 5-5 attrib object for Arm CPU components in virtually-addressed regimes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>privileged</td>
<td>Boolean</td>
<td>Access is privileged.</td>
</tr>
<tr>
<td>instruction</td>
<td>Boolean</td>
<td>For reads, access is on the instruction side if True, or data side if False.</td>
</tr>
<tr>
<td>user</td>
<td>NumberU64</td>
<td>User signaling (AXI4).</td>
</tr>
</tbody>
</table>

#### Table 5-6 attrib object for Arm CPU components in physically-addressed regimes

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonSecure</td>
<td>Boolean</td>
<td>Access is non-secure if True, or secure if False.</td>
</tr>
</tbody>
</table>
| type | String | Device or normal memory type. Must be one of the following:  
- "Device-nGnRnE" (strongly-ordered).  
- "Device-nGnRE" (device).  
- "Device-nGRE" (v8-specific).  
- "Device-GRE" (v8-specific).  
- "Normal" (innerCacheability, outerCacheability, and shareability define the attributes). |
Table 5-6  attrib object for Arm CPU components in physically-addressed regimes (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>innerCacheability</td>
<td>String</td>
<td>Cacheability for the inner domain. Must be one of:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &quot;NC&quot; (Non-Cacheable).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &quot;WT&quot; (Write-Through).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &quot;WB&quot; (Write-Back).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is only relevant for type=Normal and ignored for other types. These attributes are only used for routing the debug transaction. Debug accesses on caches have special semantics. For WT and WB there are no allocation hints for debug accesses as debug accesses never allocate. For more information, see 5.7.9 Reading and writing through caches and buffers on page 5-93.</td>
</tr>
<tr>
<td>outerCacheability</td>
<td>String</td>
<td>Cacheability for the outer domain. See innerCacheability.</td>
</tr>
<tr>
<td>shareability</td>
<td>String</td>
<td>Shareability. Must be one of:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &quot;nsh&quot; (Non-Shareable).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &quot;ish&quot; (Inner Shareable).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• &quot;osh&quot; (Outer Shareable).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is only relevant for type=Normal and is ignored for other types.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>--- Note ---</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When both innerCacheability and outerCacheability are NC, shareability is ignored and is Outer Shareable.</td>
</tr>
</tbody>
</table>

5.7.8  Reading and writing memory

The semantics of reading memory are peek rather than bus read. The semantics of writing memory are poke rather than bus write.

When reading, data is transferred in the data member of the MemoryReadResult object returned by memory_read() and when writing, data is transferred in the data argument of the memory_write() function. Both data fields use the same format to encode the data being transferred, see the documentation of memory_write() or MemoryReadResult.

Examples

These examples show how various byteWidth elements are packed into NumberU64 elements.

All these examples are correct for big-endian and little-endian target memory and for big-endian and little-endian host memory. The representation is independent of the target or host endianness.

byteWidth=1

Reading 8 bytes with values 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88:

data[0] = 0x8877665544332211

byteWidth=2

Reading 4 16-bit words with values 0x1110, 0x2120, 0x3130, 0x4140:

data[0] = 0x4140313021201110

byteWidth=4

Reading 2 32-bit words with values 0x13121110, 0x23222120:

data[0] = 0x2322212013121110
5.7.9 Reading and writing through caches and buffers

This information applies to all types of buffer, for example caches, write buffers, and temporary data buffers.

Reads through a CPU component with caches must return dirty cache data and data in write buffers, if appropriate (programmer's view). The memory component can have its own memory view, but with physical addresses. Reads never change any cache state, for example they never allocate or flush. `memory_read()` means peek rather than bus read, for non-intrusive observation. Reads of any cache or buffer must return the data that an architectural read would see. Reads usually follow the same path as architectural reads.

Writes through a CPU component with caches must write the data through to all caches and to main memory, regardless of any write through policy or allocation policy. The data, but not the metadata, of write buffers and any other temporary buffers containing memory data, whether for read or write, must be updated, even data that architecturally cannot be modified. Writes only update data, not tags or metadata. Writes never allocate and never update the dirty bit of lines. Clean data is updated everywhere, and is therefore inherently clean after the write. Dirty data is also updated everywhere and so might no longer be dirty, or in other words different, after the write.

Writes on any cache and buffer must update all known locations that hold this data. Writes are usually very different from architectural writes.

The sequence `memory_write(address=A, data=memory_read(address=A))` has the side effect of propagating the programmer's view value of address A into all caches, buffers and into main memory.

5.7.10 Address translation

The `memory_translateAddress()` function is used to translate an address in one memory space into an address in another memory space. A common example is to convert a virtual address into a physical address.

A client can get a list of useful and supported translations by calling `memory_getUsefulAddressTranslations()`. It does not necessarily return all supported translations, but all returned translations are guaranteed to be supported by `memory_translateAddress()`.

Implementing `memory_getUsefulAddressTranslations()` is optional, even when `memory_translateAddress()` is implemented.

Address translation is usually only supported for specific pairs of memory spaces and only in specific directions. If the requested translation is not supported, `E_unsupported_translation` is returned. A translation might fail even if it is supported, for example because a certain address is not mapped in the output memory space. In this case, the returned address array is empty.

Clients can derive a short and consistent description for each supported translation by using the names of the memory spaces, for example "memspace_name_in -> memspace_name_out".

5.7.11 Memory sideband information

Instances can provide sideband information for addresses in a memory space using `memory_getSidebandInfo()`. GUIs can display this information in a tooltip when the user hovers over a memory cell, for example.
The following table describes all the sideband information fields that an instance might return in `memory_getSidebandInfo()`:

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regionStart</td>
<td>NumberU64</td>
<td>The sideband information in this Object is valid for all addresses in the range <code>regionStart</code> to <code>regionEnd</code>, if they are present. However, there is no claim that this region is maximized, in other words, that it could not be further extended. Therefore, the model can return the page limits of a virtual page without looking at adjacent pages. If <code>regionStart</code> or <code>regionEnd</code> are missing, the sideband information is only valid for the requested address. The requested address is in the range <code>regionStart</code> to <code>regionEnd</code>.</td>
</tr>
<tr>
<td>regionEnd</td>
<td>NumberU64</td>
<td>End of the region for which the sideband information is valid.</td>
</tr>
<tr>
<td>physicalAddress</td>
<td>NumberU64</td>
<td>Physical address corresponding to the requested address.</td>
</tr>
<tr>
<td>ipa</td>
<td>NumberU64</td>
<td>Intermediate physical address corresponding to the requested address.</td>
</tr>
<tr>
<td>noExecute</td>
<td>Boolean</td>
<td>If True, the requested address cannot be used to execute code.</td>
</tr>
<tr>
<td>ext_&lt;Info&gt;</td>
<td>Value</td>
<td>Components can put arbitrary information here using the <code>ext_</code> prefix. Aspects that are supported consistently across multiple components can be added to this table in the future, without the <code>ext_</code> prefix.</td>
</tr>
</tbody>
</table>

### 5.7.12 memory_getMemorySpaces()

Gets a list of all memory spaces and their static information.

**Arguments**

`instId`

Type: NumberU64

Opaque number uniquely identifying the target instance.

**Return value**

`MemorySpaceInfo[]`

List of `MemorySpaceInfo` on page 5-100 Objects. Can be empty. If so, no memory can be read or written on this instance.

**Errors**

- `E_unknown_instance_id`

### 5.7.13 memory_getSidebandInfo()

Gets sideband information for a specific memory address of a specific memory space.

**Arguments**

`address`

Type: NumberU64

Addresses to get sideband information for.
attrib
Type: Map[String]Value
Optional. Transaction attributes for the memory access. See `memory_read()` on page 5-96 for details. The attributes might or might not be relevant to the returned sideband information.

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

request
Type: String[]
Optional. If present, this specifies which sideband information fields must be filled in the return value. It can contain any subset of the field names. This can be used to suppress filling in members that are expensive to produce. The instance is free to fill in more members than were requested. Unknown members are silently ignored and are not present in the result. If this is missing, all supported sideband information must be returned.

spaceId
Type: NumberU64
Opaque number uniquely identifying the memory space.

**Return value**
Map[String]Value
Object that contains the sideband information for the specified address.

**Errors**
- E_unknown_instance_id.
- E_unknown_memory_space_id.
- E_address_out_of_range.
- E_unaligned_access.
- E_unsupported_attribute_name.
- E_unsupported_attribute_value.
- E_unsupported_attribute_combination.

5.7.14 memory_getUsefulAddressTranslations()
Returns a list of useful address translations that are supported by `memory_translateAddress()`. It does not necessarily return all supported translations, but all returned translations are guaranteed to be supported by `memory_translateAddress()`.

**Arguments**
instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

**Return value**
MemorySupportedAddressTranslationResult[]
List of useful and supported address translations. See `MemorySupportedAddressTranslationResult` on page 5-102.
5.7.15 memory_read()

Reads data from the target's memory view of a given memory space. The semantics are 'peek' rather than 'bus read'.

Arguments

- **address**
  - Type: NumberU64
  - Start reading at this address. Must be within the range minAddr to maxAddr inclusive in the specified memory space, else E_address_out_of_range is returned. Must be naturally aligned according to byteWidth.

- **attrib**
  - Type: Map[String]Value
  - Optional. Transaction attributes for the memory access. The attributes are represented as key/value pairs as members in the Object. All specified attributes override the respective default attributes of the selected memory space. A subset of all supported attributes can be specified, in which case only the specified attributes are overridden.

  Specifying an empty object or omitting the attrib argument causes the default attributes of the selected memory space to be used. Specifying an unsupported or unknown attribute name results in an E_unsupported_attribute_name error. Specifying an unsupported or invalid value for a valid attribute name results in an E_unsupported_attribute_value error. Specifying an invalid combination of attributes (also with respect to default attributes that were not overridden) can result in E_unsupported_attribute_combination, but target instances are not obliged to detect all invalid attribute combinations.

  The set of attributes supported by a memory space depends on the memory space and the target instance. The MemorySpaceInfo on page 5-100 publishes all attributes and their default values in the attrib and attribDefaults fields.

- **byteWidth**
  - Type: NumberU64
  - Access width in bytes. Must be a power of two (1, 2, 4, 8 ...), else E_data_size_error is returned. Accesses that fail because of a valid but unsupported access width record the error address in the error member of the result.

- **count**
  - Type: NumberU64
  - Number of elements of byteWidth to read, starting at address and increasing the address for each element by byteWidth. Must be > 0, else E_data_size_error is returned. Models can transport accesses as bursts or individually, depending on the capabilities of the bus that is used internally.

- **instId**
  - Type: NumberU64
  - Opaque number uniquely identifying the target instance.

- **spaceId**
  - Type: NumberU64
  - Opaque number uniquely identifying the memory space.
Return value
MemoryReadResult
Object that contains the data and optional error indication for each element that was read. See MemoryReadResult on page 5-99.

Errors
• E_unknown_instance_id.
• E_unknown_memory_space_id.
• E_data_size_error.
• E_address_out_of_range.
• E_unaligned_access.
• E_unsupported_attribute_name.
• E_unsupported_attribute_value.
• E_unsupported_attribute_combination.

5.7.16 memory_translateAddress()
Translates an address of one memory space into an address of another memory space. This function is useful if the two memory spaces are associated with two different translation regimes, for example virtual and physical memory. A common example is to convert a virtual address into a physical address.

Arguments
instId
Type: NumberU64
Opaque number uniquely identifying the target instance.
address
Type: NumberU64
Input address.
outSpaceId
Type: NumberU64
Desired output memory space. The address is translated from spaceId to outSpaceId.
spaceId
Type: NumberU64
Opaque number uniquely identifying the memory space of the input address.

Return value
MemoryAddressTranslationResult
Object that contains the result of the address translation. This is either the output address or an error. See MemoryAddressTranslationResult on page 5-99.

Errors
• E_unknown_instance_id.
• E_unknown_memory_space_id.
• E_address_out_of_range.
• E_unsupported_translation.

5.7.17 memory_write()
Writes data into the target's memory view of a given memory space. The semantics are poke rather than bus write.
Arguments

address
Type: NumberU64
Start writing bytes at this address. Must be within minAddr and maxAddr of the memory space.

attrib
Type: Map[String]Value
Optional. Transaction attributes for the memory access. The attributes are represented as key/value pairs as members in the object. All specified attributes override the respective default attributes of the selected memory space. A subset of all supported attributes can be specified, in which case only the specified attributes are overridden.

Specifying an empty object or omitting the attrib argument causes the default attributes of the selected memory space to be used. Specifying an unsupported or unknown attribute name results in an E_unsupported_attribute_name error. Specifying an unsupported or invalid value for a valid attribute name results in an E_unsupported_attribute_value error. Specifying an invalid combination of attributes (also with respect to default attributes that were not overridden) can result in E_unsupported_attribute_combination, but target instances are not obliged to detect all invalid attribute combinations.

The set of attributes supported by a memory space depends on the memory space and the target instance. The MemorySpaceInfo on page 5-100 publishes all attributes and their default values in the attrib and attribDefaults fields.

byteWidth
Type: NumberU64
Access width in bytes. Must be 1, 2, 4, or 8, else E_data_size_error is returned. Accesses that fail because of a valid but unsupported access width record the error address in the error member of the result.

count
Type: NumberU64
Number of elements of byteWidth to write, starting at address and increasing the address for each element by byteWidth. Must be > 0, else E_data_size_error is returned. Models can transport accesses as bursts or individually, depending on the capabilities of the bus that is used internally.

data
Type: NumberU64[]
Data elements written to ascending addresses, packed into NumberU64 types such that the lowest address is in the lowest bits:

byteWidth=1
  8 bytes per NumberU64, lowest address in bits[7:0].

byteWidth=2
  4 x 16-bit values per NumberU64, lowest address in bits[15:0].

byteWidth=4
  2 x 32-bit values per NumberU64, lowest address in bits[31:0].

byteWidth=8
  1 NumberU64 per address.

byteWidth=16
  2 NumberU64 per address, lowest bits in the first one.
byteWidth=N

N/8 NumberU64 per address, lowest bits in the first one. Elements of byteWidth >= 2 are read with the endianness of the memory space inside each element, but elements are stored with the lowest bits inside each NumberU64 (for byteWidth < 8) and with the lowest bits first in sequences of NumberU64 (for byteWidth > 8). Elements that caused an error while writing record the error address in the error member in the result. Padding bytes (unused high bits) must be set to zero. The array must have exactly N = floor((byteWidth*count+7) / 8) NumberU64 elements (N is always >= 1).

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

spaceId
Type: NumberU64
Opaque number uniquely identifying the memory space.

Return value
MemoryWriteResult
Object that contains error information. See MemoryWriteResult on page 5-102 for details.

Errors
• E_unknown_instance_id.
• E_unknown_memory_space_id.
• E_data_size_error.
• E_address_out_of_range.
• E_unaligned_access.
• E_unsupported_attribute_name.
• E_unsupported_attribute_value.
• E_unsupported_attribute_combination.

5.7.18 MemoryAddressTranslationResult
MemoryAddressTranslationResult members:
address
Type: NumberU64[]
List of addresses in memory space outSpaceId that correspond to address in spaceId. If this array is empty then address is not mapped in outSpaceId. If the array contains exactly one element then the mapping is unique. If it contains multiple addresses then address is accessible in the same way under all of these addresses in outSpaceId.

5.7.19 MemoryReadResult
MemoryReadResult members:
data
Type: NumberU64[]
Data elements read from ascending addresses, packed into NumberU64 types such that the lowest address is in the lowest bits:

byteWidth=1
8 bytes per NumberU64, lowest address in bits[7:0].
**byteWidth=2**
4 x 16-bit values per NumberU64, lowest address in bits[15:0].

**byteWidth=4**
2 x 32-bit values per NumberU64, lowest address in bits[31:0].

**byteWidth=8**
1 NumberU64 per address.

**byteWidth=16**
2 NumberU64 per address, lowest bits in the first one.

**byteWidth=N**
N/8 NumberU64 per address, lowest bits in the first one. Elements of byteWidth >= 2 are read with the endianness of the memory space inside each element, but elements are stored without endianness inside each NumberU64 (for byteWidth < 8) and with the lowest bits first in sequences of NumberU64 (for byteWidth > 8). Elements that could not be read have a value of zero and their address is recorded in the error array. Padding bytes (unused high bits) are always zero. Array length is always exactly N = floor((byteWidth*count+7) / 8) NumberU64 elements (N is always >= 1).

**error**
Type: NumberU64[]
Optional. List of addresses and E_* error codes for memory locations that could not be read. Each entry in the list occupies two array elements: address and then errorCode. This only returns errors that happen because the callee is unable to provide the memory value, for example architectural errors. This does not return errors that are caused by the caller doing something wrong, for example E_unknown_memory_space_id. These errors are returned by memory_write() on page 5-97 instead. The errors that are most likely to occur in this list are:

**E_memory_abort**
The memory value could not be read because the memory operation was aborted by the memory subsystem. A load instruction from this address would have also failed. The corresponding bits in data must be ignored.

**E_approximation**
The memory value could be read but is only an approximation, for example because one of the instances is not in a debuggable-state. A load instruction from this address would have succeeded.

**E_value_not_available**
The memory value is currently not available and cannot even be approximated. A load instruction from this address would have succeeded. This can happen for example when an instance is not in a debuggable-state and a TLB entry is updated half-way. Addresses that failed with an error might or might not have been updated. Addresses that did not fail with an error also might or might not have been updated (for example, a read-only register that silently ignores writes, ROM).

### 5.7.20 MemorySpaceInfo

MemorySpaceInfo members:
attrib

Type: Map[String]AttributeInfo

Optional. Attributes of this memory space. Object that maps attribute names onto AttributeInfo on page 5-185 objects, which contain their type and description. The attribute names are the same as used for the attrib argument of memory_read() on page 5-96 and memory_write() on page 5-97. It is valid for a memory space and instance not to define any attributes, in which case this member might be missing. The actual set of attributes and their semantics are described in the documentation of the target instance.

attribDefaults

Type: Map[String]Value

Optional. Object that maps attribute names in this memory space to their default values. The type of the default value must be valid for memory_read() on page 5-96 and memory_write() on page 5-97 and must be consistent with the type advertised in attrib.

Some memory spaces use dynamic values for some or all attributes, for example attributes derived from the current state of a CPU. Attributes that are taken from the current dynamic state are missing in attribDefaults. Thus an empty or missing attribDefaults object indicates that all attributes are dynamic.

The value of attribDefaults can be passed directly into the attrib argument of memory_read() on page 5-96 and memory_write() on page 5-97. Doing so has the same effect as specifying no attrib argument or an empty attrib object, because this overrides the defaults with the defaults.

canonicalMsn

Type: NumberU64

Optional. Canonical memory space number. This number adheres to the scheme described by the memory.canonicalMsnScheme member returned by instance_getProperties() on page 5-182.

description

Type: String

Optional. Description of the memory space. Might contain linefeeds.

diendness

Type: String

Optional. Hint for clients about which diendness to use to interpret wider-than-byte memory values. Possible values:

little

Little-endian format (default when endianness is not present).

big

Big-endian format (also sometimes called "byte invariant big endian format").

be32

big-endian word invariant endianness

variable

The diendness can change at runtime. The client determines the current diendness by other means if it can, for example by reading a register.
The memory space does not define any endianness. Clients must use little-endianness by default and allow the user to select the endianness. Regardless of the value of the endianness field, clients must allow the user to override the endianness when displaying memory.

**maxAddr**

Type: NumberU64

Optional. Maximum address in this address space (inclusive). Default is $2^{64}$-1. Must be $\geq$ minAddr.

**minAddr**

Type: NumberU64

Optional. Minimum address in this address space (inclusive). Default is 0. It is very exotic to have a memory space with minAddr $\neq$ 0. Memory spaces must not represent blocks of memory.

**name**

Type: String

Short string identifying the memory space. If available, this is the architectural memory space name.

**spaceId**

Type: NumberU64

Opaque memory space id uniquely identifying the memory space within the target instance. Used to read and write memory locations.

### 5.7.21 MemorySupportedAddressTranslationResult

MemorySupportedAddressTranslationResult members:

**description**

Type: String

Description of this translation. This also explains artefacts, for example non-uniqueness and unsupported cases.

**inSpaceId**

Type: NumberU64

Input memory space id.

**outSpaceId**

Type: NumberU64

Output memory space id.

### 5.7.22 MemoryWriteResult

MemoryWriteResult members:

**error**

Type: NumberU64[]

Optional. List of addresses and E_* error codes for memory locations that could not be written. Each entry in the list occupies two array elements: address and then errorCode. This only returns errors that happen because the callee is unable to write the memory value, for example architectural errors. This does not return errors that are caused by the caller doing something
wrong, for example `E_unknown_memory_space_id`. These errors are returned by
`memory_read()` on page 5-96 instead. The errors that are most likely to occur in this list are:

**E_memory_abort**

The memory value could not be written because the memory operation was aborted by
the memory subsystem. A store instruction to this address would have also failed.
5.8 Disassembly API

Some instances can provide a disassembled view of their memory. Because disassembly is just a form of \texttt{memory\_read()}, a memory space id must be provided.

The main interface function is \texttt{disassembler\_getDisassembly()}, which offers a disassembled view of a memory location. The other disassembler functions offer more exotic functionality, for example querying disassembler modes and disassembling individual opcodes.

Target instances that do not support disassembly must return \texttt{E\_function\_not\_supported\_by\_instance} for the \texttt{disassembler\_\*()} functions.

\$\texttt{IRIS\_HOME/Examples/Client/Disassembly/} contains an example client application that demonstrates how to use this API.

This section contains the following subsections:

- 5.8.1 Disassembling chunks of memory on page 5-104.
- 5.8.2 Disassembling opcodes on page 5-104.
- 5.8.3 \texttt{disassembler\_disassembleOpcode()} on page 5-104.
- 5.8.4 \texttt{disassembler\_getCurrentMode()} on page 5-105.
- 5.8.5 \texttt{disassembler\_getDisassembly()} on page 5-106.
- 5.8.6 \texttt{disassembler\_getModes()} on page 5-107.
- 5.8.7 \texttt{DisassemblyLine} on page 5-107.
- 5.8.8 \texttt{DisassemblyMode} on page 5-107.

5.8.1 Disassembling chunks of memory

Use the \texttt{disassembler\_getDisassembly()} function to disassemble a chunk of memory.

This function returns lines of disassembled instructions. The number of lines that are returned is specified by the count argument. The amount of memory that a chunk represents depends on the encoding of the instruction set being disassembled. The address of the next instruction following a disassembled chunk is given by the \texttt{address} field of the last \texttt{DisassemblyLine} element of the result value. This function returns count lines, unless an error occurred.

This function can return the following errors:

- When \texttt{address} is out of the range \texttt{minAddr} to \texttt{maxAddr} for the memory space, it returns \texttt{E\_address\_out\_of\_range}.
- When reading past \texttt{maxAddr}, which is the end of the memory space, no error is returned. In this case, fewer disassembly lines than requested are returned.
- When the memory subsystem cannot read a byte value, for example due to a permission fault or a translation table fault, no error is returned. In this case, the opcode string in \texttt{DisassemblyLine} must be empty and the disassembly string must have the format "(error: foo)" where \texttt{foo} describes the error that occurred. Disassembly must continue by increasing the address by one unit of the alignment constraint until count elements are returned.

5.8.2 Disassembling opcodes

Instead of retrieving the disassembly for a specific memory location, it is possible to retrieve the disassembly for an individual opcode, using the \texttt{disassembler\_disassembleOpcode()} function.

Disassembling an opcode never fails because of an invalid opcode. Instead, \texttt{opcode=}"" and \texttt{disass=}"\texttt{hex\_constant\_definition\_in\_assembler\_syntax}" are returned in the \texttt{DisassemblyLine} object.

5.8.3 \texttt{disassembler\_disassembleOpcode()}

Retrieves the disassembly for an individual opcode.
Arguments

address
Type: NumberU64
Context for the disassembly. Address of the opcode, for example for addresses of relative branches. If unknown, specify 0.

code
Type: Object
Optional. More context information for the disassembly. The contents of this object depend on the instance and the selected mode. When this is missing, reasonable defaults are to be used.

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

mode
Type: String
Mode name to use for disassembling. This must either be one of the modes returned by disassembler_getModes() or the special mode Current which selects the mode currently returned by disassembler_getCurrentMode().

opcode
Type: NumberU64[]
Opcode to disassemble, in 64-bit units in little-endian format.

Return value
DisassemblyLine
DisassemblyLine object containing the disassembly.

Errors
• E_unknown_instance_id.
• E_unknown_disassembly_mode.

5.8.4 disassembler_getCurrentMode()

Gets a hint from the target about which disassembly mode is best suited to disassemble the memory around the current PC location of the target. It has similar semantics to resource_read(). Debuggers can use this function to implement an auto mode which always displays disassembly in the current mode.

Arguments

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

Return value
String
Name of the current mode. This is the mode that is best suited to disassemble the current PC location. If the target does not support disassembly, this is an empty string.
Errors

- E_unknown_instance_id.

5.8.5 disassembler_getDisassembly()

Disassembles a chunk of memory.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

address

Type: NumberU64

Start disassembling at this address. Must be within minAddr and maxAddr in the memory space. If the set of implemented architectures has strong per-instruction alignment requirements which are not met by address, E_unaligned_access must be returned.

attrib

Type: Object

Optional. Transaction attributes for the memory read access. The attributes are represented as key/value pairs as members in the object. See memory_read() on page 5-96 for details.

count

Type: NumberU64

Maximum number of lines to disassemble. Must be > 0.

maxAddr

Type: NumberU64

Optional. Stop disassembling at or shortly after maxAddr. One element is returned that is >= maxAddr. In any case, no more than count+1 elements are returned.

mode

Type: String

Mode name to use for disassembling. This must either be one of the modes returned by disassembler_getModes() on page 5-107 or the special mode Current which selects the mode returned by disassembler_getCurrentMode() on page 5-105.

spaceId

Type: NumberU64

Opaque number uniquely identifying the memory space.

Return value

DisassemblyLine[]

List of DisassemblyLine on page 5-107 Objects containing the disassembly. The length of this array is one element more than the number of returned disassembly lines ([0 to count] + 1). A trailing DisassemblyLine on page 5-107 element is appended which only has a valid address member containing the next address to start disassembling from. The opcode and disass members are invalid, and might be missing, for this last element.
Errors

- E_unknown_instance_id.
- E_unknown_memory_space_id.
- E_data_size_error.
- E_unknown_disassembly_mode.
- E_address_out_of_range.
- E_unaligned_access.
- E_unsupported_attribute_name.
- E_unsupported_attribute_value.
- E_unsupported_attribute_combination.

5.8.6 disassembler_getModes()

Retrieves a list of supported disassembly modes. The mode name string is used to identify the disassembly mode.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

Return value

DisassemblyMode[]

List of supported disassembly modes. Empty if the target does not support disassembly. If the target supports disassembly, this must contain at least one element. See DisassemblyMode on page 5-107.

Errors

- E_unknown_instance_id.

5.8.7 DisassemblyLine

DisassemblyLine members:

address

Type: NumberU64

Start address of this disassembled instruction.

disass

Type: String

Disassembly of this instruction. It is a single line without linefeeds, or multiple lines with linefeeds (LF, 0xa) but without a trailing linefeed. The returned string does not contain any symbols.

opcode

Type: String

Opcode of the disassembled instruction in hexadecimal format, using uppercase letters, without the leading 0x. It can contain spaces for readability. Clients are suggested to display the opcode string as returned by the target. It is a single line without linefeeds, or multiple lines with linefeeds (LF, 0xa) but without a trailing linefeed.

5.8.8 DisassemblyMode

DisassemblyMode members:
description
Type: String
Description of this disassembly mode.

name
Type: String
Short name identifying the disassembly mode.

Related references
5.7.15 memory_read() on page 5-96
5.9 Tables API

The tables interface allows an instance to expose an ordered series of records that all have the same fields.

Clients first call `table_getList()` to get a list of tables exposed by the instance. Then they call `table_read()` or `table_write()` to read or write the contents of the table cells.

This interface can be used to expose arbitrary information in tabular form. If it is more appropriate to represent the information as a resource, memory space, or disassembly, they should be used instead, because they contain semantic information.

Table rows are accessed by a densely allocated index. Each index uniquely corresponds to one table row. So, for example, index range 4-8 is 5 table rows.

Information that has a non-dense key, for example addresses, or that uses non-unique keys, for example addresses in translation tables, can expose this non-dense or non-unique key as a column and hide the index column. Then the index becomes an opaque id of a display slot.

As for resources and memory, the semantics are `peek` rather than `bus read` and `poke` rather than `bus write`, and reads and writes should be as side-effect free as possible.

This section contains the following subsections:
- 5.9.1 Table cell types on page 5-109.
- 5.9.2 `table_getList()` on page 5-109.
- 5.9.3 `table_read()` on page 5-110.
- 5.9.4 `table_write()` on page 5-110.
- 5.9.5 `TableCellError` on page 5-111.
- 5.9.6 `TableColumnInfo` on page 5-111.
- 5.9.7 `TableInfo` on page 5-113.
- 5.9.8 `TableReadResult` on page 5-114.
- 5.9.9 `TableRecord` on page 5-114.
- 5.9.10 `TableWriteResult` on page 5-114.

5.9.1 Table cell types

The following types are allowed in table cells and all clients must support them:
- `String`
- `NumberU64`
- `NumberS64`
- `Boolean`
- `NumberU64[]`

`NumberU64[]` is used to represent binary data which exceeds 64 bits, for example cache line data.

The cell values returned or specified must be consistent with the cell type specified in the `TableColumnInfo`. The `NumberU64[]` in a column has the same length for all rows. The length is specified in the `TableColumnInfo`. Clients must handle inconsistent types gracefully.

5.9.2 `table_getList()`

Gets the static meta information of all tables exposed by an instance. This information can be used to retrieve the actual volatile data and render this data in a client in a suitable form.

**Arguments**

`instId`

Type: `NumberU64`

Opaque number uniquely identifying the target instance.
5.9.3 table_read()

Reads data from a table in order to display it to the user. The semantics are peek rather than 'bus read'.
This must be as free of side effects as possible.

Arguments

- count
  - Type: NumberU64
  - Optional. Number of records to read, starting at index. Default is 1. The index index+count-1
    must be in the range minIndex to maxIndex, else E_index_out_of_range is returned.

- index
  - Type: NumberU64
  - The row number from which to start reading. This must be in the range minIndex to maxIndex,
    else E_index_out_of_range is returned.

- instId
  - Type: NumberU64
  - Opaque number uniquely identifying the target instance.

- tableId
  - Type: NumberU64
  - Opaque table id of the table to read from.

Return value

TableReadResult

List of table records that were read from the table. This is an array even if only one record was read. The
size of the array is always the same as count. See TableReadResult on page 5-114.

Errors

- E_unknown_instance_id.
- E_unknown_table_id.
- E_index_out_of_range.

5.9.4 table_write()

Writes individual fields in individual table records. The semantics are poke rather than 'bus write'. This
must be as free of side effects as possible while keeping all simulation state consistent.

Arguments

- instId
  - Type: NumberU64
  - Opaque number uniquely identifying the target instance.
records
   Type: TableRecord[]
   List of table records to be written. This is an array even if only one record is written. The size of
   the array determines how many records are written. It is valid to specify a subset of the values of
   a record to write a subset of table cells, potentially only one.

tokenId
   Type: NumberU64
   Opaque table id of the table to write to.

Return value
TableWriteResult
   An object that contains errors, if any occurred when writing cells. See TableWriteResult on page 5-114.

Errors
   • E_unknown_instance_id.
   • E_unknown_table_id.
   • E_index_out_of_range.

5.9.5 TableCellError
   TableCellError members:
   errorCode
      Type: NumberS64
      Numeric E_* error code.
   index
      Type: NumberU64
      Index in table (row number).
   name
      Type: String
      Column name (field name) as in TableColumnInfo.

5.9.6 TableColumnInfo
   TableColumnInfo members:
   description
      Type: String
      Description of the record field, describing the semantics of the record fields in free form. Can
      contain linefeeds.
   format
      Type: String
      Optional. Iris-text-format. The value of a cell is referred to by the value variable in the Iris-text-
      format string. The values of other cells are referred to by their column title, and the values of
      resources in the same instance are referred to by their resource name optionally prefixed by a
      resource group name and a dot.
formatLong
Type: String
Optional. Iris-text-format specification when displaying the value in textual form. Can contain multiple lines and must be self-descriptive. This is intended to be displayed for example in bubble help when hovering over a table cell. Must be present iff formatShort is present.

formatShort
Type: String
Optional. Iris-text-format specification when displaying the value in textual form. Ignored for type String. Must only be a single line and must be short. This is intended to be displayed in a table cell. Values are referred to by their column name. The index can be referred to by the index variable in the Iris-text-format string.

name
Type: String
Column title and record field name. This must be a short capitalized string uniquely identifying the column in the table.

rwMode
Type: String
Optional. Either "r" or "rw" for read-only or read-write (default). Clients must not try to write to read-only locations. Components must silently ignore writes to read-only locations.

bitWidth
Type: NumberU64
The interpretation of bitWidth depends on the value of type:

NumberU64, NumberS64
Size of the value in bits. The actual value is always zero-extended or sign-extended to 64 bits. This is the number of bits that are relevant to the user.

NumberU64[]
Size of the value in bits. The encoding is the same as for resources. The least significant bit is in bit[0] of the first NumberU64, the following NumberU64 values contain the more significant bits (little-endian encoding with 64-bit quantities).

Boolean
bitWidth must be 1.

String
bitWidth must be 0 and must be ignored by clients.

type
Type: String
Value type of this record field (of the table cells in this column). Must be one of the following:
• NumberU64.
• NumberS64.
• NumberU64[].
• Boolean.
• String.

Related references
5.5 Iris-text-format on page 5-64
5.9.7 TableInfo

TableInfo members:

columns
Type: TableColumnInfo[]
List of meta information for the columns in the table. This describes the fields in each record. The order in this list must be preserved by clients when displaying information.

description
Type: String
Description of the table, describing the semantics of the table in free form. Can contain linefeeds.

formatLong
Type: String
Optional. Global Iris-text-format. Long version of formatShort intended to be displayed in a bubble help when hovering over a table row. This must be specified iff formatShort is specified.

formatShort
Type: String
Optional. Global Iris-text-format. If present, clients must format the data of each row according to this format and must not (by default) show the individual columns. This short variant must be single line and is intended to be displayed in the table cells. Values are referred to by their column name. The index can be referred to by index. If missing, the table is shown as specified in columns.

indexFormatHint
Type: String
Optional. Hint for clients on how to display the index of a record. If present this must be one of the following:

hide
Hide the numeric index of records. Display the first column as row index.

dec
Display the row index preferably in decimal (but allow the user to override this).

hex
Display the row index preferably in hexadecimal (but allow the user to override this). This is useful if the index represents an address-like entity. Default is "hex".

maxIndex
Type: NumberU64
Optional. Maximum row index (inclusive). If the number of records is not known statically this must be set to \(2^{64} - 1\). It is valid to set this to very high numbers such as \(2^{64} - 1\). Clients must not blindly try to retrieve all rows of a table. Default is \(2^{64} - 1\).

minIndex
Type: NumberU64
Optional. Minimum row index (inclusive). This is usually 0. Default is 0.
name
Type: String
Name of the table. This must be a short string uniquely identifying the table in the instance.

tableId
Type: NumberU64
Opaque table id used to read and write table contents.

Related references
5.5 Iris-text-format on page 5-64

5.9.8 TableReadResult
TableReadResult members:

data
Type: TableRecord[]
The actual data read from the table.

error
Type: TableCellError[]
Optional. List of errors that occurred while reading one or more table cells. This only returns errors that happen because the callee is unable to read the table cell value of an existing table cell, for example architectural errors. This does not return errors caused by the caller doing something wrong, for example E_index_out_of_range. These errors are returned by table_read() on page 5-110 instead. This array is either missing, in case of no such error, or non-empty, in case of errors.

5.9.9 TableRecord
TableRecord members:

index
Type: NumberU64
Index of this record.

row
Type: Map[String]Value
Object with key-value pairs as described in the corresponding TableColumnInfo on page 5-111 objects.

5.9.10 TableWriteResult
TableWriteResult members:

error
Type: TableCellError[]
Optional. List of errors that occurred while writing one or more table cells. This only returns errors that happen because the callee is unable to write the table cell value of an existing table cell, for example architectural errors. This does not return errors caused by the caller doing something wrong, for example E_index_out_of_range. These errors are returned by resource_write() on page 5-76 instead. This array is either missing, in case of no such error, or non-empty, in case of errors.
5.10 Image loading and saving API

This section describes the image loading and saving interface. The \texttt{image\_load\*()} functions load images into a target instance.

In a typical implementation:

- Only target instances that have the \texttt{executesSoftware=1} property support the \texttt{image\_\*()} functions.
- Cores and CPUs support the \texttt{image\_\*()} functions.
- Memory components, for example RAMDevice, only support the \texttt{memory\_\*()} functions, not the \texttt{image\_\*()} functions.

This section contains the following subsections:

- 5.10.1 \textit{Loading an image} on page 5-115.
- 5.10.2 \textit{Saving an image} on page 5-116.
- 5.10.3 \texttt{image\_loadDataRead()} callback on page 5-116.
- 5.10.4 \texttt{image\_clearMetaInfoList()} on page 5-116.
- 5.10.5 \texttt{image\_getMetaInfoList()} on page 5-116.
- 5.10.6 \texttt{image\_loadData()} on page 5-117.
- 5.10.7 \texttt{image\_loadDataPull()} on page 5-118.
- 5.10.8 \texttt{image\_loadDataRead()} on page 5-119.
- 5.10.9 \texttt{image\_loadFile()} on page 5-120.
- 5.10.10 ImageMetaInfo on page 5-121.
- 5.10.11 ImageReadResult on page 5-121.

5.10.1 Loading an image

Select from the following functions to load an image:

\texttt{image\_loadFile()}

Loads an image into a target instance from a file. The file must be accessible under path on the host that runs the target instance. If the file is only guaranteed to be accessible on the host that runs the client, clients should use \texttt{image\_loadData()} instead, and load the file in the client.

\texttt{image\_loadData()}

Loads image data into a target instance. Clients can use this function to push an image into an instance with a single function call. This function is intended for images that are small enough to be transferred as one uninterruptible chunk, typically up to a few megabytes. To load an image from the client side that is larger than that, use \texttt{image\_loadDataPull(}).

\texttt{image\_loadDataPull()}

Loads image data into a target instance. This is semantically equivalent to \texttt{image\_loadData()}, except that the image data is not provided as an argument but is pulled by the target instance from the client by calling the callback function \texttt{image\_loadDataRead(}). This interface enables:

- Interruptible transfers of large images.
- Format loaders, for example an ELF loader, to only read specific parts of images, or to skip some data, for example debug information.
- Format loaders to read data whose size is unknown or hard to determine.

\texttt{memory\_write()}

This is the primary method of writing raw byte data into memory at a specific address.

\textbf{Note}

- The \texttt{image\_load\*()} functions can optionally load arbitrary raw binary data, which is unformatted and without a header, into memory, by using the \texttt{rawAddr} and \texttt{rawSpaceId} arguments.
- Target instances must return \texttt{E\_function\_not\_supported\_by\_instance} for any \texttt{image\_load\*()} functions that they do not support.
5.10.2 Saving an image

Use the following functions to save an image:

- `memory_read()` and `resource_read()` inspect the state of the target instance. The client is responsible for formatting this data into the required image format and writing it to a file.
- It is not possible to use the `image_*()` interface to write an image into a file that is accessible to the target instance. In other words, there is no `image_saveFile()` functionality.

5.10.3 `image_loadDataRead()` callback

The callee of `image_loadDataPull()`, the consumer, calls this function on the caller of `image_loadDataPull()`, the client, to retrieve a chunk of data. The consumer determines the read position and the size of each chunk.

Consumers should not use this function for fine-grain parsing, for example to parse a symbol table, symbol by symbol. The function call overhead should not become significant, even for IPC connections, so very small chunks should not be used. On the other hand, very large chunks should also not be used, because they block the IPC connection for the transfer of a single block. The chunk size should be between 100KB and 10MB, typically around 1MB.

On reaching the end of the file, the client returns fewer bytes than requested, possibly zero bytes, and returns `E_ok`. Reading past the end of the file is not treated as an `E_io_error`.

The client can return `E_operation_interrupted` if a user interrupted or canceled loading a large image. The implementation of `image_loadData()` should then stop calling `image_loadDataRead()` and should return `E_operation_interrupted`.

If a read error occurs, for example an error from the host OS while reading a file, the client returns `E_io_error`. Since a function can either return an error code or a result object, it does not return `ImageReadResult` nor any bytes. The implementation of `image_loadData()` should then stop calling `image_loadDataRead()` and should return `E_io_error`.

5.10.4 `image_clearMetaInfoList()`

Clears the list of meta information for the loaded images in the target instance.

**Arguments**

`instId`

Type: `NumberU64`

Opaque number uniquely identifying the target instance.

**Return value**

Function has no return value.

**Errors**

- `E_unknown_instance_id`.

5.10.5 `image_getMetaInfoList()`

Gets image filenames and other meta information as loaded by the client using `image_loadData()`, or by the target instance using `image_loadFile()`, since the list of meta information was last reset using `image_clearMetaInfoList()`.
Arguments
instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

Return value
ImageMetaInfo[]
List of image meta information records of images loaded since the last call to image_clearMetaInfoList() on page 5-116. See ImageMetaInfo on page 5-121.

Errors
• E_unknown_instance_id.

5.10.6 image_loadData()

Loads an image from a data buffer.

Loading an image might cause some state to be modified, for example the start address in the PC register, or the symbol table. This function is semantically equivalent to image_loadFile() on page 5-120 except that this function does not open a file. Instead, it expects the image contents as an argument.

Arguments
data
Type: NumberU64[]
Image data to be pushed into the target instance. If rawAddr is not specified, this byte sequence is the complete image, for example a complete ELF file. The bytes are stored in little-endian format in the NumberU64 values. The last element in the array can contain 1-8 valid bytes. Unused bytes must be set to 0 by the callee. The length of this array is floor((size+7)/8), else E_data_size_error is returned.

Only the first size bytes in data are interpreted. The remaining 0-7 unused bytes are ignored. A non-raw image file, for example an ELF file, cannot be pushed into the target in multiple chunks using this function. Use image_loadDataPull() on page 5-118 instead to achieve a chunked read of huge ELF files. Multiple ELF files can, of course, be loaded by calling image_loadData() multiple times.

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

path
Type: String
Optional. Hint to the target instance about the original client-side filename. The target instance must not use this filename to open a file. The path only makes sense on the host of the client. It might still be useful to show this path to the user, with a note that this is a path on the client side.

rawAddr
Type: NumberU64
Optional. If specified, treat the data as a contiguous chunk that starts at byte position 0 and load it to rawAddr and rawSpaceId. A format-specific loader, for example an ELF loader, is not used to load the data, even if it has a valid format signature. If not specified, the data is loaded according to its format signature, see the path argument to image_loadFile() on page 5-120.
rawSpaceId
Type: NumberU64
Optional. Memory space id to load raw data into. This must be present iff rawAddr is present, otherwise E_unknown_memory_space_id is returned.

type size
Type: NumberU64
Exact size of the image. The size of the data array is insufficient as it has a granularity of 8 bytes. The length of the data array is floor((pushDataSize+7)/8), else E_data_size_error is returned.

Return value
Function has no return value.

Errors
• E_unknown_instance_id.
• E_unknown_image_format.
• E_image_format_error.
• E_unsupported_argument_name.
• E_data_size_error.

5.10.7 image_loadDataPull()
Loads an image from a data buffer.

Loading an image might cause some state to be modified, for example the start address in the PC register, or the symbol table. This function is semantically equivalent to image_loadData() on page 5-117 except that the image data is not provided as an argument. Instead, the target instance pulls the image data from the client by calling the callback function image_loadDataRead() on page 5-119.

Arguments
instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

path
Type: String
Optional. Hint to the target instance about the original client-side filename. The target instance must not use this filename to open a file. The path only makes sense on the host of the client. It might still be useful to show this path to the user, with a note that this is a path on the client side.

rawAddr
Type: NumberU64
Optional. If specified, treat the data as a contiguous chunk that starts at byte position 0 and load it to rawAddr and rawSpaceId. A format-specific loader, for example an ELF loader, is not used to load the data, even if it has a valid format signature. If not specified, the data is loaded according to its format signature, see the path argument to image_loadFile() on page 5-120.

rawSpaceId
Type: NumberU64
Optional. Memory space id to load raw data into. This must be present iff rawAddr is present, otherwise E_unknown_memory_space_id is returned.
tag

Type: NumberU64

Opaque tag provided by the caller which is passed back to the callback function
image_loadDataRead() on page 5-119 so the caller can match the callbacks to the correct call,
like a stream id. The instance id of the caller is in the top 32 bits of the request "id" (for all
function calls).

Return value

Function has no return value.

Errors

- E_unknown_instance_id.
- E_unknown_image_format.
- E_image_format_error.
- E_unsupported_argument_name.
- E_data_size_error.
- E_operation_interrupted.

5.10.8 image_loadDataRead()

Reads and returns a chunk of data. This is the callback function of image_loadDataPull().

Arguments

tag

Type: NumberU64

This is the value of the tag argument of the image_loadDataPull() on page 5-118 call for
which this is a callback. It allows clients to match image_loadDataRead() callbacks to the
original image_loadDataPull() on page 5-118 calls.

tag

Type: NumberU64

This is the value of the tag argument of the image_loadDataPull() on page 5-118 so the caller can match the callbacks to the correct call,
like a stream id. The instance id of the caller is in the top 32 bits of the request "id" (for all
function calls).

Return value

Function has no return value.

Errors

- E_unknown_instance_id.
- E_unknown_image_format.
- E_image_format_error.
- E_unsupported_argument_name.
- E_data_size_error.
- E_operation_interrupted.

5.10.8 image_loadDataRead()

Reads and returns a chunk of data. This is the callback function of image_loadDataPull().

Arguments

tag

Type: NumberU64

This is the value of the tag argument of the image_loadDataPull() on page 5-118 so the caller can match the callbacks to the correct call,
like a stream id. The instance id of the caller is in the top 32 bits of the request "id" (for all
function calls).

Return value

Function has no return value.

Errors

- E_unknown_instance_id.
- E_unknown_image_format.
- E_image_format_error.
- E_unsupported_argument_name.
- E_data_size_error.
- E_operation_interrupted.
Return value

ImageReadResult

An object containing the chunk of data that was read. See ImageReadResult on page 5-121.

Errors

- E_unknown_instance_id.
- E_io_error.
- E_operation_interrupted.

5.10.9 image_loadFile()

Loads an image from a file. Loading an image might cause some state to be modified, for example the start address in the PC register, or the symbol table. The file must be accessible under path on the host that runs the target instance.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

path

Type: String

Path of the image file to load. It can be absolute or relative to the current working directory. This path must be on a filesystem that is accessible to the callee, but is not necessarily accessible to the client that calls the function.

Multiple image file formats are usually supported. The image file format is auto-detected, based on the content of the file. If the image format is unknown, E_unknown_image_format is returned. If the image format is known but the image could not be loaded because of unsupported content or a malformatted image, E_image_format_error is returned. The instance property loadfileExtension provides a hint to clients about which filename extensions are supported.

rawAddr

Type: NumberU64

Optional. If specified, treat the file content as a contiguous chunk of data, ignoring any format signature, and load it to rawAddr and rawSpaceId. A format-specific loader, for example an ELF loader, is not used to load the data, even if it has a valid format signature. If not specified, the file is loaded according to its format signature, see path.

rawSpaceId

Type: NumberU64

Optional. Memory space id to load raw data into. This must be present if rawAddr is present, otherwise E_unknown_memory_space_id is returned.

Return value

Function has no return value.

Errors

- E_unknown_instance_id.
- E_unknown_image_format.
- E_image_format_error.
- E_error_opening_file.
- E_io_error.
- E_unknown_memory_space_id.

**Related references**

5.20.4 Instance properties on page 5-175

### 5.10.10 ImageMetaInfo

ImageMetaInfo members:

**instId**

*Type: NumberU64*

Instance id of the instance that called the `image_load*()` function. In other words, the client that triggered loading the image. This id can be used by clients to determine whether an image was loaded by this client or by other clients.

**instanceSideFile**

*Type: Boolean*

If True the file was loaded by the target instance (using `image_LoadFile()` on page 5-120), else it was loaded by a client (using `image_LoadData()` on page 5-117). Note that different clients can run on separate hosts (separate from the target instance and separate from each other), seeing different filesystems, so it is not guaranteed that every client can see all files marked with `instanceSideFile=False`.

**path**

*Type: String*

Path passed to the `image_load*()` functions. Clients must only use this for informational purposes or for coarse loading heuristics, for example automatically loading debug information from the image when possible, as clients cannot be sure to have access to the file when running on a different host to the simulation.

**rawAddr**

*Type: NumberU64*

Optional. Base address of loaded data in memory. Only present if the image was loaded with a `rawAddr` argument.

**rawSpaceId**

*Type: NumberU64*

Optional. Only present iff `rawAddr` is present. This is the id of the memory space that `rawAddr` belongs to.

### 5.10.11 ImageReadResult

ImageReadResult members:

**data**

*Type: NumberU64[]*

Data bytes returned by the read operation. The bytes are stored in little-endian format in the `NumberU64` values of the array. The last element might contain [1-8] valid bytes. Unused bytes must be 0. The length of the array is always `floor((size+7)/8)`. 
size

Type: NumberU64

Size of the data read in bytes. size is always less than or equal to the size argument in the `image_loadDataRead()` on page 5-119 request. If it is equal to size in `image_loadDataRead()` on page 5-119 then there might be more bytes to read. If it is less than size in `image_loadDataRead()` on page 5-119 (and potentially 0) then end-of-file was encountered. In this case the caller of `image_loadDataRead()` on page 5-119 must call `image_loadDataRead(end=true, size=0)` once and then stop calling `image_loadDataRead()` on page 5-119 with this tag.
5.11 Simulation time execution control API

Simulation time execution control allows a client to stop and resume the progress of simulation time. After stopping simulation time, the client can inspect the state of the simulation. Simulation time execution control should be non-intrusive, in other words, it should not affect the behavior of the simulation.

$IRIS_HOME/Examples/Client/ExecutionControl/ contains an example client application that demonstrates how to use this API.

This section contains the following subsections:

- 5.11.1 Starting and stopping simulation time on page 5-123.
- 5.11.2 simulationTime_get() on page 5-123.
- 5.11.3 simulationTime_run() on page 5-124.
- 5.11.4 simulationTime_stop() on page 5-124.
- 5.11.5 Event source IRIS_SIMULATION_TIME_EVENT on page 5-124.

5.11.1 Starting and stopping simulation time

Clients can use simulationTime_run() and simulationTime_stop() to start and stop simulation time. These functions return asynchronously to the point in time when the simulation time starts or stops progressing. simulationTime_run() can return:

- Before the simulation time starts progressing.
- After the simulation time starts progressing.
- After the simulation time briefly started to progress, then stopped again, for example when running to a nearby breakpoint.

simulationTime_stop() can return before or after the simulation time stops progressing.

Callers must monitor the IRIS_SIMULATION_TIME_EVENT event source to find out when the simulation time starts or stops progressing.

Simulation time progresses until one of the following events happen:

- A breakpoint that stops the simulation time is hit.
- An event counter overflow occurs, with counterMode.overflowStopSim=True.
- simulationTime_stop() is called.
- An instance has executed the steps specified in step_setup().

Any kind of stepping, for example instruction stepping, is achieved by calling step_setup() followed by calling simulationTime_run().

Related references

5.13.6 step_setup() on page 5-132

5.11.2 simulationTime_get()

Gets the current simulation time in ticks and the run/stop state.

Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

Return value

SimulationTimeObject

Object containing the current simulation time in ticks, the tick resolution, and the run/stop state. See SimulationTimeObject on page 5-195.
5.11.3 simulationTime_run()

Requests to resume the progress of simulation time soon. If simulation time is already running, this function is silently ignored.

**Arguments**

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

**Return value**

Function has no return value.

5.11.4 simulationTime_stop()

Requests to stop the progress of simulation time soon. If simulation time is already stopped, this function is silently ignored.

**Arguments**

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

**Return value**

Function has no return value.

5.11.5 Event source IRIS_SIMULATION_TIME_EVENT

This event is emitted when the simulation time starts or stops progressing. It gives the reason why simulation time stopped. It is provided by the framework.SimulationEngine instance, not by individual instances.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TICKS</td>
<td>NumberU64</td>
<td>Current simulation time in ticks. One tick is 1/TICK_HZ seconds long. The elapsed simulation time is TICKS/TICK_HZ seconds.</td>
</tr>
<tr>
<td>TICK_HZ</td>
<td>NumberU64</td>
<td>Time resolution of the TICKS value in Hz. For example, 1000 means that 1 tick = 1ms.</td>
</tr>
<tr>
<td>RUNNING</td>
<td>Boolean</td>
<td>True if and only if the simulation is running, else False.</td>
</tr>
</tbody>
</table>

**Note**

This information might already be out of date when the callback is received. When multiple simulation controllers start and stop the simulation, for example if multiple debuggers are connected, there is no way to reliably know whether the simulation is currently running or stopped. In this case, this field is only a hint.
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REASON</td>
<td>NumberU64</td>
<td>Optional. This field is only present when the simulation is stopped, in other words, when RUNNING=False. It gives the reason why simulation time stopped. If there are multiple reasons, only one IRIS_SIMULATION_TIME_EVENT is generated. The reason is only a coarse classification and can usually be ignored by clients. Bit[0] UNKNOWN. Bit[1] STOP. simulationTime_stop() was called. Bit[2] BREAKPOINT. A breakpoint was hit. Details about hit breakpoints are transmitted with the instance-specific IRIS_BREAKPOINT_HIT event source. IRIS_BREAKPOINT_HIT is emitted before IRIS_SIMULATION_TIME_EVENT. Bit[3] EVENT_COUNTER_OVERFLOW. An event counter overflow occurred, with counterMode.overflowStopSim. Bit[4] STEPPING_COMPLETED. Bit[5] REACHED_DEBUGGABLE_STATE. For more information, see 5.12 Debuggable state API on page 5-126. Bit[6] EVENT. The simulation stopped because of an event in an event stream that was created with stop=True. Details about which event caused the stop are transmitted with the event callback. For example, ec_FOO for event FOO. The event callback happens before IRIS_SIMULATION_TIME_EVENT. All other bits are reserved and must be zero.</td>
</tr>
<tr>
<td>INST_ID</td>
<td>NumberU64</td>
<td>Optional. If available, this contains the instance id that originally caused the simulation time event.</td>
</tr>
</tbody>
</table>
5.12 Debuggable state API

Debuggable state is the state of an instance in which its registers, memory, and other resources can be freely inspected and manipulated.

Programmer's view simulations are usually in a debuggable state. They execute atomic transitions from one debuggable state to another. They generally do not implement the debuggable state API.

In less abstract simulation environments, for example RTL simulations, or in real hardware, a target instance might not always be in a state in which registers and memory can be freely inspected and manipulated. In such environments, instances might need to execute some simulation time to get into a debuggable state, in order to expose a consistent view of registers, memory, and other resources.

When not in a debuggable state, an instance might respond to resource reads and memory reads with approximate values and return E_approximation in the error field in ResourceReadResult and MemoryReadResult. It might be unable to provide a value at all, and return E_value_not_available. Or, it might respond with the actual value. Similarly, writes might fail when not in a debuggable state.

This section contains the following subsections:

- 5.12.1 Debuggable state flags on page 5-126.
- 5.12.2 Reaching debuggable state on page 5-127.
- 5.12.3 Testing whether a model has reached debuggable state on page 5-127.
- 5.12.4 Support for the debuggable state API on page 5-127.
- 5.12.5 debuggableState_getAcknowledge() on page 5-128.
- 5.12.6 debuggableState_setRequest() on page 5-128.
- 5.12.7 simulationTime_runUntilDebuggableState() on page 5-128.

5.12.1 Debuggable state flags

Instances that support debuggable state maintain the following flags:

**Debuggable-state-request flag**

The client sets or clears this flag in a specific instance by calling debuggableState_setRequest(). Only debuggableState_setRequest() can change this flag. The flag is per-instance and not per-client. Setting the flag changes how the instance behaves as simulation time passes in the following ways:

- If the request was not yet acknowledged, the instance progresses towards a debuggable state, for example by flushing pipelines and moving register values to their final location.
- When the request has been acknowledged, the instance is halted. It stops progressing while simulation time passes and while the request flag is set. This is necessary to be able to bring more than one instance into a debuggable state.

**Debuggable-state-acknowledge flag**

The instance automatically sets or clears this flag when it reaches or leaves a debuggable state. Typically, the instance reaches a debuggable state after setting the debuggable-state-request flag and executing some simulation time, usually by calling simulationTime_runUntilDebuggableState().

Typically, the instance leaves a debuggable state when simulation time progresses after the debuggable-state-request flag has been cleared.

The debuggableState_getAcknowledge() function queries this flag. This function has no side effects on the instance.

These flags have similar semantics to a debug request pin and a debug acknowledge pin found on some CPUs.
5.12.2 Reaching debuggable state

The global, non-instance-specific function, `simulationTime_runUntilDebuggableState()` advances simulation time until all instances for which a debuggable state is currently requested, have acknowledged it.

This function is a variant of `simulationTime_run()`. They generally have the same semantics, except that this function sets up a global state in which simulation time stops automatically when all instances that have the `debuggable-state-request` flag set also have the `debuggable-state-acknowledge` flag set.

This is a global function and so does not take an `instId` argument.

The simulation time might stop before reaching a debuggable state for the same reasons as for `simulationTime_run()`, for example hitting breakpoints, or calling `simulationTime_stop()`.

To find out whether a debuggable state was reached, enable the `IRIS_SIMULATION_TIME_EVENT` event source. The `REACHED_DEBUGGABLE_STATE` bit in the `REASON` field indicates whether a debuggable state was reached.

--- Note ---

- After requesting one or more instances enter a debuggable state, if you then call `simulationTime_run()` instead of `simulationTime_runUntilDebuggableState()`, this does not cause the simulation time to stop progressing when all the instances enter a debuggable state.
- Simulation time progresses while bringing an instance into a debuggable state, so it is intrusive. The state of an instance in the system might change as a result of bringing an instance into a debuggable state.

Related references

5.11.3 `simulationTime_run()` on page 5-124
5.11.4 `simulationTime_stop()` on page 5-124

5.12.3 Testing whether a model has reached debuggable state

Follow these steps to inspect a model that supports debuggable state:

Procedure

1. Call `debuggableState_setRequest(request = true)` on all instances to be inspected and that support the `debuggableState_setRequest()` function.
2. Call `simulationTime_runUntilDebuggableState()`.
3. Wait until an `IRIS_SIMULATION_TIME_EVENT` occurs.
4. Inspect the `REACHED_DEBUGGABLE_STATE` bit of the `REASON` field of the `IRIS_SIMULATION_TIME_EVENT` to find out whether a debuggable state was reached. If not, ignore or re-iterate depending on the desired behavior.
5. Call `debuggableState_setRequest(request = false)` on all instances for which the request was previously set to true.
6. Inspect and manipulate the state.

Related references

5.11.5 Event source `IRIS_SIMULATION_TIME_EVENT` on page 5-124

5.12.4 Support for the debuggable state API

The `debuggableState_*()` functions are typically only supported by instances that can be in a non-debuggable state.
This means:
• They are usually only supported by simulations below the programmer's view abstraction level, for example cycle-accurate simulations and RTL simulations.
• They are typically implemented by core or CPU-like instances. However, they can also be implemented by other instances that have complex behavior, for example interconnects.
• Instances that do not implement them are assumed to always be in a debuggable state.
• The global function \texttt{simulationTime\_runUntilDebuggableState()} is usually only supported if and only if there are one or more instances in the system that support the \texttt{debuggableState\_\*()} functions.

5.12.5 \texttt{debuggableState\_getAcknowledge()}

Queries the debuggable-state-acknowledge flag of an instance.

\textbf{Arguments}
\begin{itemize}
  \item \texttt{instId}
\end{itemize}

\texttt{Type: NumberU64}

Opaque number uniquely identifying the target instance.

\textbf{Return value}
\begin{itemize}
  \item \texttt{Boolean}
\end{itemize}

Returns true iff the instance is in a debuggable state, else false.

\textbf{Errors}
\begin{itemize}
  \item \texttt{E\_unknown\_instance\_id}.
\end{itemize}

5.12.6 \texttt{debuggableState\_setRequest()}

Sets or clears the debuggable-state-request flag in a specific instance. This flag is changed only by this function, and does not change spontaneously.

\textbf{Arguments}
\begin{itemize}
  \item \texttt{instId}
\end{itemize}

\texttt{Type: NumberU64}

Opaque number uniquely identifying the target instance.

\begin{itemize}
  \item \texttt{request}
\end{itemize}

\texttt{Type: Boolean}

Default: True

Optional. If missing or true, set the debuggable-state-request flag, that is, request to go into a debuggable state. If present and false, clear the debuggable-state-request flag, that is, resume normal execution. Default is true.

\textbf{Return value}
\begin{itemize}
  \item Function has no return value.
\end{itemize}

\textbf{Errors}
\begin{itemize}
  \item \texttt{E\_unknown\_instance\_id}.
\end{itemize}

5.12.7 \texttt{simulationTime\_runUntilDebuggableState()}

A variant of \texttt{simulationTime\_run()} and generally has the same semantics, except that this function sets up a global state in which the progress of simulation time stops automatically when all instances that have the debuggable-state-request flag set also have the debuggable-state-acknowledge flag set.
Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

Return value

Function has no return value.

Related references

6.1 Function-specific error codes on page 6-203
5.7.19 MemoryReadResult on page 5-99
5.6.19 ResourceReadResult on page 5-84
5.13 Stepping API

The step_*() functions allow you to progress the global simulation time so that a specific instance advances by a number of steps. They only set or query state that is used for stepping. They do not resume simulation time.

Stepping is non-intrusive. In other words, the execution result is the same whether code is freely running or being stepped through.

Note

Stepping does not just advance the state of the stepped instance, it advances the global simulation time for all instances.

When the specified instance has executed the specified number of steps, the progress of simulation time is stopped and an IRIS_SIMULATION_TIME_EVENT with the STEPPING_COMPLETED bit in the REASON field is generated.

This section contains the following subsections:

- 5.13.1 Step units on page 5-130.
- 5.13.2 Step counters on page 5-130.
- 5.13.3 Examples on page 5-131.
- 5.13.4 step_getRemainingSteps() on page 5-131.
- 5.13.5 step_getStepCounterValue() on page 5-132.
- 5.13.6 step_setup() on page 5-132.

5.13.1 Step units

The step_*() functions use a generic concept of steps.

The meaning of a step is defined by the unit argument, which can have the following values:

- unit = "instruction"
  One step corresponds to one executed instruction. This unit should be supported by all types of CPU models, regardless of the simulation technology.

- unit = "cycle"
  One step corresponds to one executed clock cycle of the target instance. Not all models support this value.

5.13.2 Step counters

Each instance that supports stepping maintains the following counters:

- A global step counter. This counts the steps in an increasing and wrapping 64-bit counter.
- A global remaining steps counter. This counts down the remaining steps during stepping. If it transitions from one to zero, the simulation is stopped. If it is zero, no stepping is performed. The remaining steps counter of an instance is shared by all clients. Therefore, only one connected client can step an instance.

When the simulation time stops, for any reason, the remaining steps counter of all instances is automatically set to zero to disable stepping. For example, this might happen when:

- A breakpoint is hit.
- A stepping operation of another instance has completed.
- simulationTime_stop() was called.

Related references

5.11.4 simulationTime_stop() on page 5-124
5.13.3 Examples

To step an instance, a client first calls `step_setup()` on it to set the remaining steps counter to N steps. This function does not execute any steps and does not start or resume the progress of simulation time.

To execute N steps of a specific instance, the progress of simulation time needs to be resumed explicitly, for example with the following function calls:

```python
step_setup(instId=<instId>, steps=<N>, unit="instruction");
simulationTime_run();
```

Stepping only one core instance, or a subset of core instances is achieved by combining stepping with per-instance execution control, see 5.14 Per-instance execution control API on page 5-134. The following example progresses one specific core in a system that contains multiple cores, by N steps, using per-instance execution control:

```python
step_setup(instId=<instId>, steps=<N>, unit="instruction");
perInstanceExecution_setStateAll(instanceSet=[<instId>], enable = True);
simulationTime_run();
```

Note: The order of `step_setup()` and `perInstanceExecution_setStateAll()` does not matter, because they only set state.

If the simulation stops for any reason before the specified number of steps have been executed, for example because a breakpoint is hit or `simulationTime_stop()` was called, the remaining steps count is cleared and resuming the simulation time does not resume the stepping operation, instead it freely runs the simulation.

Related references
5.11.3 `simulationTime_run()` on page 5-124
5.11.4 `simulationTime_stop()` on page 5-124

5.13.4 `step_getRemainingSteps()`

Queries the current 'remaining steps' counter value of an instance. A value of zero means that the instance is not stepping, but is running freely.

Arguments

- `instId`
  - Type: NumberU64
  - Opaque number uniquely identifying the target instance.

- `unit`
  - Type: String
  - Unit of the return value. See `unit` argument of `step_setup()` on page 5-132.

Return value

- NumberU64
  - Number of remaining steps to execute before stopping the simulation time. If zero, this instance is currently not stepping in this unit.

Errors

- E_unknown_instance_id
- E_unit_not_supported
5.13.5  step_getStepCounterValue()

Queries the current ‘step counter’ value of an instance in the specified units.

This can be used to query and monitor the current instruction count or cycle count of an instance. The step counter is a 64-bit unsigned value that is always increasing and silently wrapping around, so it must always be assumed to have an unknown offset. It can still be used to measure relative step counts.

**Arguments**

- **instId**
  Type: NumberU64
  Opaque number uniquely identifying the target instance.

- **unit**
  Type: String
  Unit of the return value. See unit argument of *step_setup()* on page 5-132.

**Return value**

NumberU64

Number of total elapsed steps in this instance so far. This wraps around and so always has an unknown offset. It is still useful to get relative readings.

**Errors**

- E_unknown_instance_id.
- E_unit_not_supported.

5.13.6  step_setup()

Sets the 'remaining steps' counter to N steps for a specified instance. Steps are measured in the specified units.

**Arguments**

- **instId**
  Type: NumberU64
  Opaque number uniquely identifying the target instance.

- **steps**
  Type: NumberU64
  Number of steps to perform before stopping the simulation time. This overwrites the previous 'remaining steps' counter of the instance. A value of zero cancels a currently running stepping operation.

  **Note**
  To run the simulation time, *simulationTime_run()* on page 5-124 must be called after calling this function.

- **unit**
  Type: String
  Unit for steps. Must be one of:

  **instruction**
  A step is one executed instruction. Typically, all core models support this unit.
cycle

A step is one cycle. Not all core models support this unit.

Return value

Function has no return value.

Errors

• E_unknown_instance_id.
• E_unit_not_supported.

Related references

5.11.5 Event source IRIS_SIMULATION_TIME_EVENT on page 5-124
5.14 **Per-instance execution control API**

Per-instance execution control allows clients to enable or disable execution of individual targets. The per-instance execution state is maintained in each instance and is shared by all callers and clients.

Execution of an instance can only progress while the simulation time of the whole system progresses. Therefore, the per-instance execution control cannot progress the state of individual components when the simulation time is stopped.

To achieve the effect of progressing execution of a single instance only, the per-instance execution of all other instances that support this feature is disabled and then the simulation time is progressed, either by free running it or by letting it run to a breakpoint.

Updating the per-instance execution state of one or more instances while the simulation time is running is allowed, but has undefined results, because the instances might detect the state change after an undefined delay. Updating the per-instance execution state is only guaranteed to work deterministically while the simulation time is stopped.

$IRIS_HOME/Examples/Client/ExecutionControl/ contains an example client application that demonstrates how to use this API.

This section contains the following subsections:
- 5.14.1 **perInstanceExecution_getState()** on page 5-134.
- 5.14.2 **perInstanceExecution_getStateAll()** on page 5-134.
- 5.14.3 **perInstanceExecution_setState()** on page 5-135.
- 5.14.4 **perInstanceExecution_setStateAll()** on page 5-135.

5.14.1 **perInstanceExecution_getState()**

Gets the current per-instance execution state of a specific instance.

**Arguments**

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

**Return value**

Object

Object containing one member, "enable" (Boolean), which is True iff execution is enabled in this instance.

**Errors**

- E_unknown_instance_id.

5.14.2 **perInstanceExecution_getStateAll()**

Gets the execution state of all instances in the system that support per-instance execution control. This is a convenience function, calling perInstanceExecution_getState() on all instances in the system that support it.

**Return value**

Object

Object containing two members: enabledSet and disabledSet. Each member is an array of NumberU64 containing the set of instance ids for which execution is enabled or disabled, respectively.
5.14.3 \texttt{perInstanceExecution\_setState()}

Gets the current per-instance execution state of a specific instance.

**Arguments**

\texttt{enable}

Type: \texttt{Boolean}

Iff True, enable execution of instructions or processing of work items.

\texttt{instId}

Type: \texttt{NumberU64}

Opaque number uniquely identifying the target instance.

**Return value**

Function has no return value.

**Errors**

- \texttt{E\_unknown\_instance\_id}.

5.14.4 \texttt{perInstanceExecution\_setStateAll()}

Updates the execution state of all instances in the system that support per-instance execution control. This is a convenience function, calling \texttt{perInstanceExecution\_setState()} on all instances in the system that support it.

**Arguments**

\texttt{enable}

Type: \texttt{Boolean}

Iff True, enable execution of instructions or processing of work items for instances in \texttt{instanceSet} and disable execution for all other instances. Iff False, disable execution for instances in \texttt{instanceSet} and enable execution for all other instances.

\texttt{instanceSet}

Type: \texttt{NumberU64[]} 

List of instance ids for which execution must be enabled or disabled, depending on \texttt{enable}. All instances that are not in this list are set to the opposite state. Can be empty.

**Return value**

Function has no return value.

**Errors**

- \texttt{E\_unknown\_instance\_id}.  

---  

\texttt{Copyright © 2018–2020 Arm Limited or its affiliates. All rights reserved.}  
\texttt{Non-Confidential}
5.15  **Breakpoints API**

Clients manipulate breakpoints in an instance by using the `breakpoint_set()` and `breakpoint_delete()` functions.

Clients are encouraged to use the `breakpoint_getList()` function to display all breakpoints, instead of maintaining their own list of breakpoints. This ensures that breakpoints that are set by other clients are visible to the user, and avoids the program stopping on breakpoints that are invisible and undeletable by the user. The use of `breakpoint_getAdditionalConditions()` is exotic.

--- Note ---

The term *breakpoint* is used here to refer to all types of breakpoints, including watchpoints and trigger points, which stop the entire simulation. When debugging an application that is running on an OS in a simulation, you usually would not use these breakpoints. This use case normally requires the simulation, and therefore the simulated OS, to continue running. Instead, you would use a debug server in the simulated world to stop the execution of the simulated application only. Iris does not support this type of application debugging.

Breakpoints and breakpoint ids are specific to the target instance that contains them, not to the client that sets them.

The target instance implementation must ensure that after a breakpoint is hit, stopping the simulation time, it is not immediately hit again when resuming the simulation time, for example by implementing a micro step.

Instances that support the breakpoint interface must also support the event interface. This allows clients to enable IRIS_BREAKPOINT_HIT events for an instance.

Debug accesses do not trigger breakpoints.

An IRIS_STATE_CHANGED event is generated when a breakpoint list changes.

--- Note ---

$IRIS_HOME/Examples/Client/Breakpoints/ contains an example client application that demonstrates how to use this API.

This section contains the following subsections:

- 5.15.1 Breakpoint actions and trace points on page 5-136.
- 5.15.2 Other ways to stop simulation time on page 5-137.
- 5.15.3 breakpoint_delete() on page 5-137.
- 5.15.4 breakpoint_getAdditionalConditions() on page 5-138.
- 5.15.5 breakpoint_getList() on page 5-138.
- 5.15.6 breakpoint_set() on page 5-138.
- 5.15.7 BreakpointAction on page 5-141.
- 5.15.8 BreakpointConditionInfo on page 5-141.
- 5.15.9 BreakpointInfo on page 5-141.
- 5.15.10 Event source IRIS_BREAKPOINT_HIT on page 5-142.

### 5.15.1  **Breakpoint actions and trace points**

Optionally, a single action can automatically be performed when a breakpoint is hit, using the *action* argument of `breakpoint_set()`. This action is performed before the `IRIS_BREAKPOINT_HIT` callback is called.

Breakpoint actions are useful for breakpoints that do not stop the simulation time, which is controlled by the *dontStop* argument of `breakpoint_set()`. The action is performed without involving the client, which reduces latency.
The action argument of `breakpoint_set()` supports the following values:

**action=eventStream_enable**
Enable the event stream esId. Ignored if the event stream is already enabled. The event stream esId must have been created before this breakpoint is set.

**action=eventStream_disable**
Disable the specified event stream esId. Ignored if the event stream is already disabled.

These two actions can be used to implement *trace points*. For more information, see *Arm Development Studio User Guide* on the Arm Developer website.

Only a single action is supported for each breakpoint, but you can set multiple identical breakpoints with different actions. The actions are executed in an undefined order.

Breakpoints can trigger arbitrary actions by implementing them in the client in the `IRIS_BREAKPOINT_HIT` callback. To execute the action synchronously with the breakpoint hit, set the breakpoint with `syncEc=true` to achieve a synchronous `IRIS_BREAKPOINT_HIT` callback. Depending on the frequency of the breakpoint hit events, this might severely affect performance, especially for clients connected using IPC.

### 5.15.2 Other ways to stop simulation time

In addition to `IRIS_BREAKPOINT_HIT` events, there are other ways to stop simulation time:

**Event breakpoints**
Events in enabled event streams that were created with `eventStream_create(stop=True)` stop the simulation time whenever they are generated. These stopping events do not generate `IRIS_BREAKPOINT_HIT` events, but they do generate event callbacks, see 5.17.3 Event callback functions on page 5-148.

**Stepping**
The stepping functionality stops the simulation time after a specified number of steps, see 5.13 Stepping API on page 5-130.

**Debuggable state**
The debuggable state functionality allows clients to stop simulation time when one or more instances reaches a debuggable state, see 5.12 Debuggable state API on page 5-126.

### 5.15.3 breakpoint_delete()

Deletes the specified breakpoint.

**Arguments**

**bptId**
Type: `NumberU64`
Breakpoint id of the breakpoint to delete.

**instId**
Type: `NumberU64`
Opaque number uniquely identifying the target instance.

**Return value**
Function has no return value.

**Errors**
- `E_unknown_instance_id`
- `E_unknown_breakpoint_id`
- `E_error_deleting_breakpoint`
5.15.4 breakpoint_getAdditionalConditions()

Discovers any component-specific breakpoint conditions that are supported by an instance.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

type

Type: String

Optional. If present, return only conditions that are applicable to the specified breakpoint type. Must be one of: "code", "data", or "register". See breakpoint_set() on page 5-138.

Return value

BreakpointConditionInfo[]

List of BreakpointConditionInfo on page 5-141 objects indicating additional conditions that can be configured.

Errors

• E_unknown_instance_id.
• E_unsupported_breakpoint_type.

5.15.5 breakpoint_getList()

Gets information about one or all breakpoints that have been set.

Arguments

bptId

Type: NumberU64

Optional. If specified, just return the information for the specified breakpoint. If not specified, return the information for all breakpoints.

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

Return value

BreakpointInfo[]

List of BreakpointInfo on page 5-141 Objects that contain information about one or all breakpoints.

Errors

• E_unknown_instance_id.
• E_unknown_breakpoint_id.

5.15.6 breakpoint_set()

Sets and adds a breakpoint on a specific instance.
Arguments

**action**

Type: `BreakpointAction`

Optional. Perform the specified action whenever the breakpoint is hit. There are actions defined to enable or disable event streams. See *BreakpointAction* on page 5-141 Object.

**address**

Type: `NumberU64`

Optional. Address or start address of a range for code and data breakpoints. Mandatory for code and data breakpoints. Ignored for other breakpoint types.

**conditions**

Type: `Map[String,Value]`

Optional. Key-value pairs specifying additional component-specific breakpoint conditions. See *breakpoint_getAdditionalConditions()* on page 5-138.

**dontStop**

Type: `Boolean`

Optional. If present and true, do not stop the simulation on breakpoint hit, but still generate an `IRIS_BREAKPOINT_HIT` event with field `DONTSTOP=true`. This feature is used to implement trace trigger points. Together with `syncEc=True`, this can be used to check for arbitrary conditions in the client before stopping the simulation, or for causing other effects at runtime, for example enabling or disabling trace, or writing resources.

**noCallback**

Type: `Boolean`

Optional. If present and true, do not generate an `IRIS_BREAKPOINT_HIT` event when the breakpoint is hit. A breakpoint action and simulation stop is still executed regardless of this option. This feature is useful for breakpoints that have an `eventStream_insertTrigger` action to avoid generating unnecessary `IRIS_BREAKPOINT_HIT` callbacks.

**instId**

Type: `NumberU64`

Opaque number uniquely identifying the target instance.

**rscId**

Type: `NumberU64`

Optional. Resource id for register breakpoints. Mandatory for register breakpoints, ignored for other breakpoint types.

**rwMode**

Type: `String`

Default: `rw`

Optional. Either "r", "w", or "rw" for read-only, write-only, or read-write (default). Only relevant for data and register breakpoints, ignored for other breakpoint types.
size

Type: NumberU64

Optional. Size of address range in bytes for code and data breakpoints. Ignored for other breakpoint types. The default is 0 which means to hit when the access or execution addresses match exactly. For size > 0, breakpoints hit when the access or execution is in the inclusive range [address to address+size-1].

spaceId

Type: NumberU64

Optional. Memory space id for code and data breakpoints. Mandatory for code and data breakpoints, ignored for other breakpoint types.

cyncEc

Type: Boolean

Default: False

Optional. If present and true, call the IRIS_BREAKPOINT_HIT callback synchronously for this breakpoint. If this is true this overrides the syncEc setting given in eventStream_create(IRIS_BREAKPOINT_HIT) on page 5-150. This allows breakpoint-hit callbacks to be generally asynchronous and the synchronous behavior can be switched on for specific breakpoints only. The default is false. See syncEc argument of eventStream_create() on page 5-150 function.

type

Type: String

Breakpoint type. Must be one of code, data, or register:

"code" (Disassembly breakpoint)

Hit just before the instruction at address in memory space spaceId is executed. When size is specified and > 0, hit just before any instruction in the range [address to address+size-1] is executed.

"data" (Watchpoint)

Hit while or shortly after address in memory space spaceId was accessed with an access matching rwMode. When size is specified and > 0, hit while or shortly after an address in the range [address to address+size-1] was accessed. Setting a data breakpoint implicitly requests a syncLevel of 2 (POST_INSN_IO).

register

Hit while or shortly after register rsId was accessed with an access matching rwMode. If the type of breakpoint is not supported by the instance, E_unsupported_breakpoint_type is returned.

Return value

NumberU64

Breakpoint id (bptId). This is specific to the instance.

Errors

• E_unknown_instance_id.
• E_unsupported_argument_combination.
• E_invalid_rwMode.
• E_invalid_breakpoint_condition.
• E_error_setting_breakpoint.
5.15.7  **BreakpointAction**

BreakpointAction members:

---

**action**

Type: String

Action that is executed when the breakpoint is hit. It might be:

- 'eventStream_enable': Enable the event stream specified by `esId`.
- 'eventStream_disable': Disable the event stream specified by `esId`.
- 'eventStream_*': Any other action starting with 'eventStream_' supported by the event stream specified by `esId`.

---

**esId**

Type: NumberU64

Optional. Event stream id for eventStream_* actions.

5.15.8  **BreakpointConditionInfo**

BreakpointConditionInfo members:

---

**name**

Type: String

Name of the condition.

---

**type**

Type: String

Type of the condition value, for example String or NumberU64.

---

**description**

Type: String

A description of the condition. This indicates the circumstances that cause the breakpoint to trigger.

---

**bptTypes**

Type: String[]

Optional. List of breakpoint types ("code", "data", or "register") that this condition is applicable to. If omitted, all types are supported.

5.15.9  **BreakpointInfo**

BreakpointInfo members:

---

**bptId**

Type: NumberU64

Breakpoint id.

---

**type**

Type: String

Breakpoint type. Is one of "code", "data", or "register".
address
Type: NumberU64
Optional. Address or start address of the range for code and data breakpoints.

size
Type: NumberU64
Optional. Size of address range in bytes for code and data breakpoints. Ignored for register
breakpoints. The default is 0 which means to hit when the access or execution address match
exactly. For size > 0, breakpoints hit when the access or execution is in the inclusive range
[address to address+size-1].

spaceId
Type: NumberU64
Optional. Memory space id for code and data breakpoints.

rscId
Type: NumberU64
Optional. Resource id for register breakpoints.

drwMode
Type: String
Optional. Either "r", "w", or "rw" for read-only, write-only or read-write (default). Only relevant
for data and register breakpoints.

dontStop
Type: Boolean
Optional. Iff present and true, do not stop simulation on breakpoint hit, but still generate an
IRIS_BREAKPOINT_HIT event with field DONTSTOP=true.

noCallback
Type: Boolean
Optional. Iff present and true, do not generate an IRIS_BREAKPOINT_HIT event when the
breakpoint is hit. A breakpoint action and simulation stop is still executed regardless of this
option. This feature is useful for breakpoints that have a eventStream_insertTrigger action to
avoid generating unnecessary IRIS_BREAKPOINT_HIT callbacks.

instId
Type: NumberU64
Instance id of the instance that created the breakpoint.

conditions
Type: Object
Optional. Key-value pairs indicating values of any additional conditions set on this breakpoint.

5.15.10 Event source IRIS_BREAKPOINT_HIT
This event is generated whenever a breakpoint is hit.

Breakpoints can only be hit while the simulation time is running. Normally the simulation time is
stopped when a breakpoint is hit, unless dontStop=true was set on breakpoint_set(). In this case, the
IRIS_BREAKPOINT_HIT event is generated but the simulation time continues running.
To receive this event from an instance, clients must explicitly call `eventStream_create(IRIS_BREAKPOINT_HIT)` once on that instance.

Multiple breakpoints might be hit before the simulation time is stopped. It is guaranteed that all `IRIS_BREAKPOINT_HIT` callbacks are called, and therefore all breakpoint hit information is present, before `IRIS_SIMULATION_TIME_EVENT(running=False)` is called.

This event source is instance-specific, unlike `IRIS_SIMULATION_TIME_EVENT`, which is global.

### Table 5-9 Event source IRIS_BREAKPOINT_HIT

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPT_ID</td>
<td>NumberU64</td>
<td>Breakpoint id of the breakpoint that was hit.</td>
</tr>
<tr>
<td>PC</td>
<td>NumberU64</td>
<td>PC value when the breakpoint was hit.</td>
</tr>
<tr>
<td>PC_SPACE_ID</td>
<td>NumberU64</td>
<td>Memory space id of the PC value when the breakpoint was hit.</td>
</tr>
<tr>
<td>ACCESS_ADDR</td>
<td>NumberU64</td>
<td>Optional. Address of the access that hit a data breakpoint. Mandatory for data breakpoints. Not present for other breakpoint types.</td>
</tr>
<tr>
<td>ACCESS_SIZE</td>
<td>NumberU64</td>
<td>Optional. Size in bytes of the access that hit a data breakpoint. Mandatory for data breakpoints. Not present for other breakpoint types.</td>
</tr>
<tr>
<td>ACCESS_RW</td>
<td>String</td>
<td>Optional. Either &quot;r&quot; or &quot;w&quot;. The rwMode of the access that hit a data or register breakpoint. Mandatory for these breakpoint types. Not present for other breakpoint types.</td>
</tr>
<tr>
<td>ACCESS_DATA</td>
<td>NumberU64[]</td>
<td>Optional. Transferred read data or write data of the access that hit a data or register breakpoint. Mandatory for these breakpoint types. Not present for other breakpoint types.</td>
</tr>
<tr>
<td>TYPE</td>
<td>String</td>
<td>Breakpoint type. One of:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• register</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See <code>breakpoint_set()</code>.</td>
</tr>
<tr>
<td>DONTSTOP</td>
<td>Boolean</td>
<td>Optional. If and only if present and True, the simulation time did not stop because of this breakpoint hit, although it might be stopped because of other breakpoints that were hit.</td>
</tr>
</tbody>
</table>

### Related references

5.11.5 Event source IRIS_SIMULATION_TIME_EVENT on page 5-124

### Related references

5.16.1 Event source IRIS_STATE_CHANGED on page 5-144
5.16 Notification and discovery of state changes API

Clients can observe the state of component instances and might display this state to the user, so they need to know when the state is updated.

When a client calls a state-modifying function like `resource_write()` or `memory_write()` an `IRIS_STATE_CHANGED` event is automatically generated. The client does not need to generate it.

All clients register for `IRIS_SIMULATION_TIME_EVENT`. When this event is received with a value of `RUNNING=False`, this indicates that the simulation time has stopped progressing. Clients should then query all displayed state and update their views.

While the simulation time is stopped, clients can modify the state of component instances, for example by writing resources or memory. Other clients should register for `IRIS_STATE_CHANGED` events to get notification of these changes and update their views.

5.16.1 Event source IRIS_STATE_CHANGED

An instance generates this event whenever its state is modified by another instance. Clients should re-read all relevant state from instances because it might have changed.

`IRIS_STATE_CHANGED` events are generated automatically when any of the following state is changed by an Iris function:

- Resource values. Resources are registers or parameters. See 5.6.14 `resource_write()` on page 5-76.
- Memory contents. See 5.7.17 `memory_write()` on page 5-97 and 5.10 Image loading and saving API on page 5-115.
- Table contents. See 5.9.4 `table_write()` on page 5-110.
- Sync level. See 5.19.4 `syncLevel_request()` on page 5-170 and 5.19.3 `syncLevel_release()` on page 5-169.

This event is not guaranteed to be generated after changes to the following:

- List of registered instances. See 5.20.29 Event source IRIS_INSTANCE_REGISTRY_CHANGED on page 5-188 for this change.
- Active event streams. This should be transparent to other instances, by definition.
- Run or stop state of the simulation time. See 5.11.5 Event source IRIS_SIMULATION_TIME_EVENT on page 5-124 for this change.

`IRIS_STATE_CHANGED` events are hints that state might have changed. They might be sent even if no change happened.

`IRIS_STATE_CHANGED` has no fields. It does not provide information about what changed or in which instances because it is generally impossible to determine this information accurately. For example, a memory write might change arbitrary registers in the same instance or in other instances, or a resource write might change anything in the same instance or in other instances, including memory, tables, and the sync level.

The `IRIS_STATE_CHANGED` event is generated with a slight delay after the last state change was observed by the global instance. The maximum delay for a single state change is about 500ms and the maximum rate is about 1 event every 500ms. The purpose of the delay is to prevent multiple events from occurring in a short period of time. Clients are usually only interested in the final state change in a set of changes, so they can update their views once, after all changes have finished.

Related references

5.11.5 Event source IRIS_SIMULATION_TIME_EVENT on page 5-124
5.6.14 `resource_write()` on page 5-76
5.7.17 `memory_write()` on page 5-97
5.17 Events and trace API

All Iris events are handled through this interface.

Target instances can expose zero or more event sources. Event sources emit:

- Trace events, for example INST events for each executed instruction.
- Simulation events, for example IRIS_BREAKPOINT_HIT events.
- Other events, for example events defined by a custom GUI.

An instance that produces events is called an event producer, or trace producer. It implements the event_*( ) functions.

An instance that receives event callbacks is called an event consumer, or trace consumer. It must implement the callbacks for the events it requested, for example ec_INST( ) and ec_IRIS_BREAKPOINT_HIT( ).

Instances that do not expose any event sources must either return E_function_not_supported_by_instance for the event_*( ) functions, or return an empty list of event sources. In practice, however, all instances that implement Iris interfaces that might generate events, must also implement the event interface to expose them, and allow clients to receive them. In particular, instances that support the breakpoint interface or the semihosting interface must implement the event interface.

Most clients only need to call the following functions:

- event_getEventSources( )
  Get a list of all events that an instance supports.

- eventStream_create( )
  Enable receiving the specified event.

- eventStream_destroy( )
  When the events are no longer required.

Alternatively, clients can enable or disable events by calling eventStream_enable( ) or eventStream_disable( ). These functions might provide a performance benefit over eventStream_create( ) and eventStream_destroy( ) when repeatedly enabling and disabling the same event stream.

The client implements the event callback functions ec_*( ), as needed.

The other functions in this API are more unusual and deal with ringbuffering events, counter events, and reading the state of an event.

$IRIS_HOME/Examples/Client/SimpleTraceClient/ contains an example client application that demonstrates how to use this API.

This section contains the following subsections:

- 5.17.1 References in event source fields on page 5-146.
- 5.17.2 Creating and destroying event streams on page 5-147.
- 5.17.3 Event callback functions on page 5-148.
- 5.17.4 Event counter on page 5-148.
- 5.17.5 ec_FOO( ) on page 5-148.
- 5.17.6 eventStream_action( ) on page 5-149.
- 5.17.7 eventStream_create( ) on page 5-150.
- 5.17.8 eventStream_destroy( ) on page 5-152.
- 5.17.9 eventStream_disable( ) on page 5-152.
- 5.17.10 eventStream_enable( ) on page 5-153.
- 5.17.11 eventStream_flush( ) on page 5-153.
- 5.17.12 eventStream_getCounter( ) on page 5-154.
- 5.17.13 eventStream_getState( ) on page 5-154.
- 5.17.14 eventStream_setOptions( ) on page 5-155.
5.17.1 References in event source fields

Event sources must follow these rules when referring to fields or to resources.

Identifying resources and memory spaces

Event sources that refer to a resource or to a memory space should use the numeric resource id, ResourceInfo.rscId or MemorySpaceInfo.spaceId, not the resource name string. If resource names are reported, which is discouraged, they must be consistent with the ResourceInfo.name or MemorySpaceInfo.name field.

Syntax of references in the format string

A format string can refer to values in this and other instances, with the following syntax and semantics. In the following list, variables are enclosed by angle brackets, <…>, and optional items are enclosed by square brackets, [...]. This list is ordered roughly from more common to less common references:

%{<field_name>…}

Refers to a field in this event source. The client must make sure the field is enabled.

%{:resource.[<group_name>].<resource_name>…}

Refers to a resource in this instance. Only well-defined for synchronous event sources, otherwise the results are undefined.

%{:event.<event_source_name>.<event_field>…}

Refers to a field in another, previous, event. The client must make sure the other event and field are enabled and must buffer the values of these fields. The semantics are only well-defined for event sources with a defined ordering relationship. Because event sources can only refer to past events, not future events, usually only the last event in a causal event chain can refer to all fields of that event chain. Clients are not expected to wait until data is available. Using this reference requires detailed knowledge of the ordering of the event sources involved.

%{:instance.<instance_path>:resource.[<group_name>].<resource_name>…}

Refers to a resource in another instance. This is only well-defined for synchronous events where other instances are stable and are causally connected to this instance. For example an event source from a downstream cache might be able to refer to resources in its parent cache or parent core if they are causally connected. In some cases, an approximation of otherwise hard to produce data might still be useful, for example acquiring the approximate PC value of an otherwise unrelated sibling core to debug multithreading problems. Using this reference requires detailed knowledge of the causal relationship between instances and their events.

%{:instance.<instance_path>:event.<event_source_name>.<event_field>…}

Refers to an event field in a previous event of another instance. The client must make sure the event and field are enabled in the other instance and the client must buffer the values of these fields. This is only well-defined if the events involved have a strict ordering relationship and if the instances involved are strongly causally connected. This might be the case, for example, when transactions flow through multiple components of a memory subsystem. Using this reference requires very detailed knowledge about the causal relationship between instances and their events.
<instance_path>
This is always relative to the instance that contains the format string. The semantics are similar to C++ namespace scoping semantics. It starts at the path of the instance that contains the format string, tries to find instance_path, and if that fails, repeatedly goes one level up, until it is found. This enables references to children, siblings, and parent instances without needing to specify absolute paths. The special token "PARENT" can be used as an explicit inverse scope to reach parents without specifying the parent's name. It can be specified multiple times, but only at the beginning of the path.

5.17.2 Creating and destroying event streams

eventStream_create() creates a new event stream, which is identified by an event stream id, esId. Event streams are destroyed by eventStream_destroy() or by calling instanceRegistry_unregisterInstance() either explicitly or implicitly.

By default, eventStream_create() enables event generation for the selected event. The ec_<eventName> callback is called on the instance specified by ecInstId, which is usually the client creating the event stream, with the requested event source fields, or a superset of them.

There are internal checks and states associated with an event stream that might suppress event generation:

• Enable flag. An event stream can be enabled or disabled. This state is controlled by:
  — The disable argument of eventStream_create().
  — The functions eventStream_enable() and eventStream_disable().
  — Trace point breakpoints. These are breakpoints with action=eventStream_enable or eventStream_disable, see the BreakpointAction object.

• Range check. An enabled event stream might only emit the events that match the ranges for a specific event source field, or the PC. See eventStream_setTraceRanges().

• Latency. eventStream_create() and eventStream_destroy() do not take effect until the next sync point, in other words, the point at which a stop can be detected. Therefore, event generation might be delayed after calling eventStream_create(), depending on the sync level of the event-producing instance and on the nature of the events, for example INST events, which are generated by instruction execution. Stopping event generation might also be delayed after calling eventStream_destroy(). For more information, see 5.19 Simulation accuracy (sync levels) API on page 5-167.

To stop generating events for a specific event stream that was previously started with eventStream_create(), a client can call either:

• eventStream_destroy(). The event stream id, esId, is no longer valid after entering this function.
• instanceRegistry_unregisterInstance(). This unregisters instance X and automatically destroys all event streams that were sent to instance X, that is, event streams that were created using eventStream_create(ecInstId=X).

Use the following guidelines on whether to use eventStream_create() and eventStream_destroy() or eventStream_enable() and eventStream_disable():

• Use eventStream_create() and eventStream_destroy() to enable and disable events interactively, at a low frequency. These functions might have some latency until events are generated, depending on the current sync level and the nature of the events. Only eventStream_destroy() ensures all event generation overhead is removed. Use these functions if you can, or if you are unsure.

• Use eventStream_enable() and eventStream_disable() to enable and disable event generation at a high frequency, for example while the simulation is running. Trace points are an example of this. These functions usually have the lowest possible latency until event generation starts or stops. A disabled event stream might have a significant runtime overhead, depending on the frequency of the event. Use these functions only if you must.

Related references

5.17.15 eventStream_setTraceRanges() on page 5-155
5.17.3 Event callback functions

Event callback functions are called for each event in an event stream that was previously created and enabled by `eventStream_create()`.

The function name `ec_FOO()` is used as a placeholder for the real callback function name. The real name is specified with the `ecFunc` argument of the `eventStream_create()` call. If the `ecFunc` argument was not specified, then the function name is `ec_<EventSourceInfo.name>`, for example `ec_INST` for the INST event source.

When calling `eventStream_create()`, if `syncEc` is not specified or False, which is usually the case, the callback function is called asynchronously to the thread causing the event. If `syncEc` is True, the callback is called synchronously. This blocks the calling thread from executing in the target instance.

5.17.4 Event counter

To count events in an event stream without the runtime overhead of using event callbacks, set the optional `counter` argument of the `eventStream_create()` function to True.

This argument has the following effects:
- An internal counter is incremented for each event. This counter is specific to an event stream and client. In other words, it is specific to each `eventStream_create()` call.
- No normal `ec_FOO()` callbacks are generated, but see `counterMode.nonOverflowTrace`.

The counter is initially set to the `startVal` argument that is passed to `eventStream_create()`, or to zero if `startVal` is not specified. Together with the `counterMode` argument, `startVal`, which is of type `NumberU64`, can be used to trigger an action after `N` events by setting it to `0xffffffffffffffff-N+1`.

The `counterMode` argument determines what happens when the counter overflows from `0xffffffffffffffff` to 0 and what happens on non-overflow events. Several orthogonal actions can be selected by setting flags in `counterMode`, see `EventCounterMode` for details.

In addition to causing actions on counter overflow and counter events, the counter value of a counting event stream can be read by using `eventStream_getCounter()`.

In the counter mode that is created using `eventStream_create(counter=True)`, the counter counts from 0 to $2^{64}$-1 and then automatically wraps to 0. This overflow can safely be ignored in all cases. The difference between two 64-bit counter values is the number of ticks between the counter values if fewer than $2^{64}$ ticks occurred between reading the counter values.

For `counterMode.overflowReload`, clients must also enable `counterMode.overflowTrace` to determine the number of wraparounds that occurred.

5.17.5 `ec_FOO()`

Event callback function.

**Arguments**

- `instId`
  - Type: `NumberU64`
  - Opaque number uniquely identifying the target instance to which this event is sent. This is not the source of the event, see `sInstId`. This argument can safely be ignored by the client (callee). This argument is used to route the events to the correct receiving instance.

- `esId`
  - Type: `NumberU64`
  - Event stream id.
fields
Type: Map[String]Value
Object that contains the names and values of all event source fields requested by the eventStream_create() on page 5-150 call. This can be a superset of fields requested by the eventStream_create() on page 5-150 call. It is guaranteed to contain all fields requested by eventStream_create() on page 5-150. Counter overflow events have an additional implicit overflowTrace field in the fields array in the ec_FOO() callback, with value True, to clearly identify this callback as an overflow event.

sInstId
Type: NumberU64
Source instId. The instance that generated and sent this event.

syncEc
Type: Boolean
Default: False
Optional. Synchronous callback behavior. If this is present and True, this invocation of ec_FOO() blocks the execution of the thread that generated the event. Otherwise the execution of the thread that generated the event resumes before this ec_FOO() call returned. This is only True iff it was set to True in eventStream_create() on page 5-150.

time
Type: NumberU64
Simulation time timestamp of the event. This is local simulation time, if available, using the terminology of temporal decoupling, or otherwise the best approximation of simulation time available to the instance. The units of this integer timestamp are called ticks and are defined by the simulation environment. The semantics are identical to simulationTime_get() on page 5-123.ticks. Instances must report a timestamp. If they are not able to produce a timestamp, they must set this to simulationTime_get() on page 5-123.ticks when generating the event.

Return value
Function has no return value.

5.17.6 eventStream_action()
Execute an action on the event stream. This is called by breakpoint actions with a BreakpointAction object and is normally not called directly by clients.

Arguments

instId
Type: NumberU64
Optional. Opaque number uniquely identifying the target instance.

esId
Type: NumberU64
Id of the event stream to execute the action on. This is the return value of eventStream_create().
**action**

Type: BreakpointAction

Object specifying the action to be executed.

**Return value**

Function has no return value.

**Errors**

- E_unknown_instance_id.
- E_unknown_event_stream_id.
- E_not_supported_for_event_source.

### 5.17.7 eventStream_create()

Creates a new event stream and returns its esId.

**Arguments**

- **instId**
  
  Type: NumberU64
  
  Optional. Opaque number uniquely identifying the target instance.

- **counter**
  
  Type: Boolean
  
  Optional. Iff True, this event stream counts events and does not emit any ec_FOO() on page 5-148 callbacks for events. It can emit counter overflow events. See also the startVal and counterMode arguments. If the event source does not support counter mode, E_counter_mode_not_supported is returned.

- **counterMode**
  
  Type: EventCounterMode
  
  Optional. Only relevant if 'counter==True'. Defines what happens when the counter overflows from 0xffffffff to 0 and what happens on non-overflow events.

- **disable**
  
  Type: Boolean
  
  Optional. Iff present and True, do not enable event generation. Event generation can later be enabled with eventStream_enable() on page 5-153 or with breakpoints that have 'action=eventStream_enable'. If this argument is missing or False, event generation is enabled.

- **ecFunc**
  
  Type: String
  
  Optional. Name of the callback function to call on instance ecInstId. This must start with ec, for 'event callback'. See ec_FOO() on page 5-148 for a description of this callback function. The default when not specified is ec_<EventSourceInfo.name>, for example ec_INST for the INST event source. Multiple event streams can send their events to the same callback function. The callback function can demultiplex events based on the event stream id. The client implementation must decide which events are sent to which function.

- **ecInstId**
  
  Type: NumberU64
  
  Callback instId. Events are sent to this instance whenever they occur.
**evSrcId**

Type: NumberU64

Id of the event source to start tracing. This must be one of the EventSourceInfo.evSrcId on page 5-158 values returned by event_getEventSources() on page 5-157.

**fields**

Type: String[]

Optional. List of requested event source fields. The array can be empty, in which case only the event is reported. Omit this argument to conveniently select all event fields. An event producer must generate all the requested fields, and can generate a superset of them, but it must avoid generating unrequested fields that are expensive to generate.

**startVal**

Type: NumberU64

Optional. Only relevant if 'counter==True'. Set the start value of the counter. Default is zero. This is useful to trigger overflow events after N events by setting startVal to -N. This is silently ignored if 'counter==False'.

**stop**

Type: Boolean

Optional. Iff present and True, stop the simulation time for each event while the event stream is enabled. The simulation time is not stopped while the event stream is disabled. An IRIS_SIMULATION_TIME_EVENT is generated for each stopping event. This option implements event breakpoints.

**syncEc**

Type: Boolean

Optional. Synchronous callback behavior. If this is present and True, the event callback function ec_FOO() on page 5-148 blocks the execution of the calling thread in the target instance until the callback function returns. Synchronous callbacks have a performance impact and must only be enabled when necessary. Synchronous callbacks enabled by out-of-process clients over IPC have a disastrous performance impact due to network latencies.

If this is missing or False, the event callback function ec_FOO() on page 5-148 might be queued and called from another thread to the simulation thread that encountered the event. In any case, the execution of the calling thread in the target instance resumes before the callback returns. This mode must be used for all normal tracing activity, which usually does not require the synchronous callback behavior.

**options**

Type: Map[String]Value

Optional. Object specifying event source-specific options for the event stream. Only specific event sources support options. The supported options (if any) are described in 'EventSourceInfo.options' of the event source. This is exotic. 'E_unsupported_option_name', 'E_unsupported_option_value' or 'E_unsupported_option_combination' is returned if a specified option is unknown, its value is unsupported or the combination of options is not supported, respectively.

**Return value**

NumberU64
Event stream id (esId). This id uniquely identifies the event stream created by this call. It is used to manipulate and stop this event stream. Trace streams are specific to a given ecInstId. Each client has its own private event stream ids.

**Errors**
- E_unknown_instance_id.
- E_unknown_event_source_id.
- E_unknown_event_field.
- E_unknown_callback_instance_id.
- E_unsupported_option_name.
- E_unsupported_option_value.
- E_unsupported_option_combination.
- E_syncEc_mode_not_supported.
- E_counter_mode_not_supported.
- E_error_creating_event_stream.
- E_error_enabling_event_stream.

**5.17.8 eventStream_destroy()**

Stops generating events for a specific event stream and destroys the event stream.

**Arguments**

- instId
  - Type: NumberU64
  - Optional. Opaque number uniquely identifying the target instance.

- esId
  - Type: NumberU64
  - Id of the event stream to be destroyed. This is the return value of eventStream_create() on page 5-150.

**Return value**

Function has no return value.

**Errors**
- E_unknown_instance_id.
- E_unknown_event_stream_id.
- E_error_destroying_event_stream.

**5.17.9 eventStream_disable()**

Disables an existing event stream. If the event stream is already disabled, this function is silently ignored. The event stream is not destroyed and can be re-enabled later by calling eventStream_enable().

**Arguments**

- instId
  - Type: NumberU64
  - Optional. Opaque number uniquely identifying the target instance.

- esId
  - Type: NumberU64
  - Id of the event stream to be enabled. This is the return value of eventStream_create() on page 5-150.
Return value
Function has no return value.

Errors
• E_unknown_instance_id.
• E_unknown_event_stream_id.

5.17.10 eventStream_enable()
Enables an existing event stream. If the event stream is already enabled, this function is silently ignored.

Arguments
instId
Type: NumberU64
Optional. Opaque number uniquely identifying the target instance.
esId
Type: NumberU64
Id of the event stream to be enabled. This is the return value of eventStream_create() on page 5-150.

Return value
Function has no return value.

Errors
• E_unknown_instance_id.
• E_unknown_event_stream_id.

5.17.11 eventStream_flush()
Flushes any potentially buffered data for an event stream by emitting all pending data via an event callback.

This is an exotic function since most event streams issue events as they happen, unbuffered. This is only meaningful and only implemented for event streams that buffer data, or that emit events in a buffered way. Event callbacks generated by an eventStream_flush() call get passed an additional field FLUSH_REQUEST_ID which contains the request id of the eventStream_flush() call that caused the event callback. Note that the event callback will generally happen asynchronously to the eventStream_flush() callback. This function might also be called while the event stream is disabled and will then also generate an event callback.

Arguments
instId
Type: NumberU64
Optional. Opaque number uniquely identifying the target instance.
esId
Type: NumberU64
Id of the event stream to be flushed. This is the return value of eventStream_create().

Return value
Function has no return value.
Errors
• E_unknown_instance_id.
• E_unknown_event_stream_id.
• E_not_supported_for_event_source.

5.17.12 eventStream_getCounter()
Gets the current event counter value for a specific event stream.

Arguments
instId
Type: NumberU64
Opaque number uniquely identifying the target instance.
esId
Type: NumberU64
Event stream id of the counter to be returned. The event stream must have been started with counter=True. Trace streams with counter=False return E_not_a_counter.

Return value
NumberU64
Current counter value of the requested event stream id.

Errors
• E_unknown_instance_id.
• E_unknown_event_stream_id.
• E_not_a_counter.

5.17.13 eventStream_getState()
Queries the current state of the resource associated with an event stream.

Arguments
instId
Type: NumberU64
Optional. Opaque number uniquely identifying the target instance.
esId
Type: NumberU64
Id of the event stream to be queried. This is the return value of eventStream_create() on page 5-150.

Return value
EventState
Object that contains the fields of the event. The fields represent the current state of the resource. See EventState on page 5-159.

Errors
• E_unknown_instance_id.
• E_unknown_event_stream_id.
• E_not_supported_for_event_source.
### 5.17.14 eventStream_setOptions()

Set options for an event stream. This is used to change the options of an existing event stream. This is an exotic function since most event streams do not support any options.

**Arguments**

- **instId**
  
  Type: NumberU64
  
  Optional. Opaque number uniquely identifying the target instance.

- **esId**
  
  Type: NumberU64
  
  Id of the event stream to set the options on. This is the return value of eventStream_create().

- **options**
  
  Type: Map[String]Value
  
  Object specifying event source specific options for the event stream. Only specific event sources support options. The supported options (if any) are described in 'EventSourceInfo.options' of the event source. This is exotic. 'E_unsupported_option_name', 'E_unsupported_option_value' or 'E_unsupported_option_combination' is returned if a specified option is unknown, its value is unsupported, or the combination of options is not supported, respectively.

**Return value**

Function has no return value.

**Errors**

- E_unknown_instance_id.
- E_unknown_event_stream_id.
- E_not_supported_for_event_source.
- E_unsupported_option_name.
- E_unsupported_option_value.
- E_unsupported_option_combination.

### 5.17.15 eventStream_setTraceRanges()

Sets a number of ranges and masks/values for an event stream. Events are only generated when a specific aspect, for example the PC of a core, matches any of the ranges or mask/value tuples.

**Arguments**

- **aspect**
  
  Type: String
  
  Can be any event source field name, or 'pc':

  - pc
    
    Enable tracing if the associated PC is in one of the masked ranges. The PC does not have to be, but can be, a field of the event sources.

  - any field name
    
    Enable event generation if the value of the field matches any of the masked ranges. The field must have been enabled in the fields argument of eventStream_create() on page 5-150. If aspect is not supported for the selected event source, E_not_supported_for_event_source is returned. If the field was not enabled for the selected event stream, E_unknown_event_field is returned.
esId
Type: NumberU64
Event stream id to be gated by the trace ranges.

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

ranges
Type: NumberU64[]
List of (start, end, mask) tuples. The array length must be dividable by 3. Tracing is enabled iff
(general case):
(start & mask) <= (aspect & mask) <= (end & mask)
Standard case: Match against [start, end]:
(start, end, 2^64-1)
Standard case: Match against mask/value:
(value, value, mask)
An empty array is valid and enables tracing permanently, as if no trace range was specified.
Event generation is enabled if one of the specified ranges matches.

Return value
Function has no return value.

Errors
• E_unknown_instance_id.
• E_unknown_event_stream_id.
• E_not_supported_for_event_source.
• E_unknown_event_field.

5.17.16  event_getEventSource()
Gets information about an event source by its name. This function also returns hidden event sources, if
their name is known to the caller.

Arguments

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

name
Type: String
Name of the requested event source.

Return value
EventSourceInfo
Metadata about the requested event source. See EventSourceInfo on page 5-158.
Errors
• E_unknown_instance_id.
• E_unknown_event_source.

5.17.17 event_getEventSources()
Retrieves a list of all event sources that are supported by a target instance. This function does not return
any hidden event sources. Use event_getEventSource() for this.

Arguments
instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

Return value
EventSourceInfo[
Zero or more EventSourceInfo on page 5-158 Objects containing the metadata for all event sources
offered by this instance.

Errors
• E_unknown_instance_id.

5.17.18 EventCounterMode
EventCounterMode members:
nonOverflowTrace
Type: Boolean
Optional. Iff present and True, call ec_FOO() on page 5-148 callback for all non-overflow
events. For example, this feature can be used to automatically trace the next 100 events. If
nonOverflowTrace and overflowTrace are both set, all events are emitted similarly to normal
non-counter event sources.

overflowDisableTrace
Type: Boolean
Optional. Iff present and True, disable event on overflow. The event stream is not destroyed. The
client must call eventStream_destroy() on page 5-152 to destroy the event stream.

overflowReload
Type: Boolean
Optional. Iff present and True, set internal counter value to startVal on overflow. If this is
False or missing, the counter wraps from 0xffffffff to 0.

overflowStopSim
Type: Boolean
Optional. Iff present and True, stop simulation on overflow.

overflowTrace
Type: Boolean
Optional. Iff present and True, call ec_FOO() on page 5-148 callback on overflow. The fields
argument has an additional implicit overflowTrace field (with value True) to clearly identify
this callback as an overflow event.
5.17.19 EventSourceFieldInfo

EventSourceFieldInfo members:

description
Type: String
Optional. Description of the event source field. Can contain linefeeds.

enums
Type: EnumElementInfo[]
Optional. Array of EnumElementInfo on page 5-186 objects which describe symbols for numeric event field values. Debuggers can display these symbols (and potentially their description) in addition to the numeric value.

name
Type: String
Name of the event source field. This name is used to uniquely identify the event source field within an event source. By convention, all field names are uppercase and can contain underscores. Field names must not start with a colon. Names starting with a colon are reserved.

size
Type: NumberU64
Size of the field value in bytes. Permissible sizes depend on the type, see type. 0 means variable size (for "type=string" and "type=uint[]" only).

type
Type: String
Type of this event source field:

uint
Hint that a value must be interpreted as an unsigned int (size must be 1, 2, 4, or 8).

int
Hint that a value must be interpreted as a signed int (size must be 1, 2, 4, or 8).

bool
Hint that a value must be interpreted as a bool (size must be 1).

float
Hint that the byte pattern must be interpreted as an IEEE 754 floating-point value (size must be 4 or 8).

string
The value is (must be) a string of variable size (size must be 0). Hint that a value must be interpreted as text.

uint[]
The value is (must be) an array of NumberU64. This is used to represent binary data. If variable length data that is not always a multiple of 8 bytes needs to be represented an explicit SIZE field must be present since the exact size cannot be derived from the length of the array.

Use int or uint together with enums to model an enum type.

5.17.20 EventSourceInfo

EventSourceInfo members:
counter

Type: Boolean

Optional. If present and True, this event source supports the counter features, see arguments counter, startVal, and counterMode for `eventStream_create()` on page 5-150, and the counter argument for `eventStream_create()` on page 5-150, if present, can be True or False. If this is False or missing, the counter argument for `eventStream_create()` on page 5-150 must be False or missing.

description

Type: String

Optional. Description of the event source. Can contain linefeeds.

evSrcId

Type: NumberU64

Opaque event source id. This id is used to identify the event source in other event functions.

fields

Type: `EventSourceFieldInfo[]`

Metadata for the fields of this event. Event sources that do not have any fields, which is rare, return an empty array.

format

Type: String

Optional. Iris-text-format. The format can refer to fields of this event source, to resources of this and other instances, and to the fields in other event sources of this and other instances.

name

Type: String

Name of the event source. This name is used to uniquely identify the event source within a target instance. By convention, all event source names are uppercase and can contain underscores. Event source names must be chosen so that wildcard matching is possible. In particular, generic parts of event source names like _START and _END must be suffixes. Alphabetic sorting of event source names must group related event sources together, if possible. Event source names must not start with a colon. Names starting with a colon are reserved.

options

Type: `Map[String]AttributeInfo`

Optional. Object mapping the names of all available event source-specific options (passed in the options argument of `eventStream_create()`) onto `AttributeInfo` on page 5-185 objects. The `AttributeInfo` on page 5-185 objects specify details of each option like type, description, and enum values. This is only present for event sources that accept options, which is exotic.

**Related references**

5.5 Iris-text-format on page 5-64

5.17.21 EventState

EventState members:

esId

Type: NumberU64

Event stream id.
fields
Type: Object
Same semantics as \texttt{ec\_FOO(fields)} on page 5-148, except that the values of the fields represent the current state of the underlying resource and no actual event happened.

time
Type: NumberU64
Same semantics as \texttt{ec\_FOO(timestamp)} on page 5-148.
sInstId
Type: NumberU64
Same semantics as \texttt{ec\_FOO(sInstId)} on page 5-148.

\subsection*{5.17.22 TraceEventData}

TraceEventData members:

\texttt{esId}
Type: NumberU64
Event stream id.

\texttt{fields}
Type: Object
Object that contains the names and values of all event source fields requested by the \texttt{eventStream\_create()} on page 5-150 call. See \texttt{ec\_FOO()} on page 5-148.

\texttt{sInstId}
Type: NumberU64
Source instId. The instance that generated and sent this event.

\texttt{time}
Type: NumberU64
Simulation time timestamp of the event. See \texttt{ec\_FOO()} on page 5-148.

\textit{Related references}

\textit{5.15.10 Event source IRIS\_BREAKPOINT\_HIT} on page 5-142
5.18 **Semihosting API**

Iris provides basic support for stdin, stdout, and stderr. It also supports the addition of new semihosting functions and the replacement of the existing ones.

Clients enable semihosting by creating event streams for the IRIS_SEMIHOSTING_* events. The semihosting_*() functions implemented by the target instance provide dedicated feedback from the client to the target instance and are usually only called from within the callback function implemented by the client. Only semihosting_provideInputData() can be called from outside of callback functions.

$IRIS_HOME/Examples/Client/Semihosting/ contains an example client application that demonstrates how to use this API.

This section contains the following subsections:

- 5.18.1 Basic stdin, stdout, and stderr support on page 5-161.
- 5.18.2 Enabling semihosting input on page 5-162.
- 5.18.3 Extending or replacing the semihosting implementation on page 5-163.
- 5.18.4 semihosting_notImplemented() on page 5-163.
- 5.18.5 semihosting_provideInputData() on page 5-163.
- 5.18.6 semihosting_return() on page 5-164.
- 5.18.7 Event source IRIS_SEMIHOSTING_OUTPUT on page 5-164.
- 5.18.8 Event source IRIS_SEMIHOSTING_INPUT_REQUEST on page 5-165.
- 5.18.9 Event source IRIS_SEMIHOSTING_INPUT_UNBLOCKED on page 5-165.
- 5.18.10 Event source IRIS_SEMIHOSTING_CALL on page 5-165.
- 5.18.11 Event source IRIS_SEMIHOSTING_CALL_EXTENSION on page 5-166.

### 5.18.1 Basic stdin, stdout, and stderr support

Clients can capture semihosting output through the stdout and stderr file descriptors, and semihosting applications can read input from stdin.

**Semihosting output through stdout and stderr**

Semihosting output is implemented as an event source, IRIS_SEMIHOSTING_OUTPUT. To receive semihosting output, clients must activate this event source using eventStream_create(). Multiple clients can request semihosting output at the same time. All of them receive the same semihosting output.

If no client requests semihosting output, the global instance must either print all semihosting output to the simulation process's host stdout file descriptor, or make it visible through another mechanism. If a client has requested semihosting output, the global instance must not print semihosting output to stdout.

Target instances that do not support any semihosting output must not expose an IRIS_SEMIHOSTING_OUTPUT event source.

**Semihosting input through stdin**

Semihosting input is more complex than output because it requires more cooperation between the user, the client, and the simulated application. The process of receiving semihosting input generally involves the following steps:

1. The simulated application tells the semihosting interface that it wants to receive user input.
2. The simulated application waits for semihosting input, either because the read call is blocked or because it actively waits.
3. The user enters data.
4. The user tells the user interface that data entry is complete.
5. The client provides the data to the simulated application.
6. The simulated application tells the semihosting interface that it is no longer waiting for semihosting input.

Target instances that do not support any semihosting input must not expose the IRIS_SEMIHOSTING_INPUT_* event sources. The implementation of semihosting_provideInputData() must return E_function_not_supported_by_instance.
The simulation and all interfaces involved must stay responsive during semihosting input. Semihosting input must not change the simulation state. A simulator that is blocked in a semihosting input operation is still considered to be running if it was running previously. It can be stopped and resumed when in this state.

Related references
5.17.7 eventStream_create() on page 5-150

5.18.2 Enabling semihosting input

Enabling semihosting input typically involves the following steps:

Procedure
1. The client activates the IRIS_SEMIHOSTING_INPUT_REQUEST event source and optionally IRIS_SEMIHOSTING_INPUT_UNBLOCKED using eventStream_create(). This step tells the global instance that this client is able to provide semihosting input. If multiple clients activate these event sources, they all receive these events. The input that they provide might be interleaved and unsynchronized.
2. When the application requires user input, it causes the semihosting implementation to issue an IRIS_SEMIHOSTING_INPUT_REQUEST event. The client might then read from the console or open a terminal window UI, for example. Applications that do not require input do not issue these events.
3. When the user has entered a character or a line of data, the client passes this data to the semihosting interface using the function semihosting_provideInputData().
4. If the simulated application requires more data, the client waits for user input and sends it to the semihosting interface when it is available. If the application does not require more data, for example it is no longer blocked in a read() call, the semihosting implementation issues an IRIS_SEMIHOSTING_INPUT_UNBLOCKED event. The client can then close the terminal window or simply ignore this event.
5. The client can send user input to the semihosting implementation using semihosting_provideInputData() at any time, even if the application is not waiting for data, in other words before an IRIS_SEMIHOSTING_INPUT_REQUEST or after an IRIS_SEMIHOSTING_INPUT_UNBLOCKED event. In these cases, the semihosting implementation must buffer the data for subsequent reads. Sending user input to the model before the first IRIS_SEMIHOSTING_INPUT_REQUEST must assume that the IRIS_SEMIHOSTING_INPUT_REQUEST_RAW field is False. This means user input should be passed to the model when the user presses the Enter key, in other words terminal cooked mode.

The following rules apply to buffering the data that is provided by semihosting_provideInputData(). They are only relevant when the semihosting implementation must handle megabytes of input data, for example from streams or files. For interactive user input, they are not relevant and can be ignored:

- The target must not impose a limit on the buffer.
- The client is responsible for not pushing too much data to the model before the model has processed it. The client should send a maximum of 1MB of data to the simulation before receiving an IRIS_SEMIHOSTING_INPUT_REQUEST event. After receiving this event, the client can send a maximum of SIZE_HINT + 1MB of data, because this event indicates that all previous data has been consumed. This ensures that the model only needs to buffer at most 1MB in addition to the currently requested data size, assuming only a single client is pushing data. The client effectively controls the buffer size in the model. The advantage over the model limiting the data size is that the client can choose a suitable behavior when more data becomes available. For example, it can block the thread that produces the data, or it can discard unwanted data.

Related references
5.17.7 eventStream_create() on page 5-150
5.18.3 Extending or replacing the semihosting implementation

Targets can contain a semihosting implementation that provides libc functionality. Applications can use this functionality without a simulated operating system.

Such an implementation can be enabled using suitable CPU parameters. No external interface is involved in making it work. However, the concept of offloading work from the simulation to the host is generic and can be used beyond the libc set of functions.

Functions that need to return more data than an integer should write it directly into memory using `memory_write()`. A buffer consisting of a memory address and length is usually passed to the called function as an argument for this purpose.

**Related references**

5.7.17 `memory_write()` on page 5-97

5.18.4 `semihosting_notImplemented()`

If a client has registered the `IRIS_SEMIHOSTING_CALL_EXTENSION` event source but only wants to override a subset of the semihosting functions, it can call `semihosting_notImplemented()` from within the `ec_FOO()` callback function for `IRIS_SEMIHOSTING_CALL_EXTENSION`.

This indicates that the function was not yet executed and that it must be handled by another client or by the original built-in function. This function is called in the same context as, and instead of, `semihosting_return()` on page 5-164.

**Arguments**

`instId`

Type: `NumberU64`

Opaque number uniquely identifying the target instance. This must be the `instId` argument of the current `ec_FOO()` on page 5-148 callback.

**Return value**

Function has no return value.

**Errors**

- `E_unknown_instance_id`.
- `E_invalid_context`.

5.18.5 `semihosting_provideInputData()`

Feeds user input as a sequence of bytes into the model for a semihosted read.

**Arguments**

`data`

Type: `NumberU64[]`

Characters (bytes) to be provided to the file descriptor `fDes` as input data. This must not be empty. The encoding is 8 bytes per array element. The first byte is in the lowest bits (little-endian). The last element contains 1-8 bytes in the lowest bits.

`fDes`

Type: `NumberU64`

File descriptor number used for input. This is usually 0, meaning stdin. This value must be passed to the `semihosting_provideInputData()` function when passing data.
instId

Type: NumberU64

Opaque number uniquely identifying the target instance to which the input data must be sent.

size

Type: NumberU64

Number of bytes in data.

Return value

Function has no return value.

Errors

- E_unknown_instance_id.
- E_unknown_file_descriptor.
- E_data_size_error.

5.18.6 `semihosting_return()`

Returns a value from a semihosting override function. This function must be called from within the event callback of the `IRIS_SEMIHOSTING_CALL_EXTENSION` event, before returning from the callback. It must not be called from any other event callback. If called from outside of an `IRIS_SEMIHOSTING_CALL_EXTENSION` event, `E_invalid_context` is returned.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the target instance to which the return value must be sent. This must be the instId argument of the current `ec_FOO()` on page 5-148 callback.

retval

Type: NumberU64

Return value of the function being called. The semantics depend on the function being called.

Return value

Function has no return value.

Errors

- E_unknown_instance_id.
- E_invalid_context.

5.18.7 Event source `IRIS_SEMIHOSTING_OUTPUT`

This event is generated by the target instance to emit bytes of semihosting output.
Table 5-10  Event source IRIS_SEMIHOSTING_OUTPUT

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>NumberU64[]</td>
<td>Characters (bytes) that are to be written to stdout or stderr. It can include bytes with the value of zero, and is not zero-terminated. It must not be empty. The encoding is 8 bytes for each array element, with the first byte in the lowest bits, in other words, little-endian. The last element contains 1-8 bytes in the lowest bits.</td>
</tr>
<tr>
<td>SIZE</td>
<td>NumberU64</td>
<td>Number of bytes in DATA.</td>
</tr>
<tr>
<td>FDES</td>
<td>NumberU64</td>
<td>File descriptor number that is used for output. 1 means stdout and 2 means stderr. Other values have a meaning that is specific to the target instance or target application.</td>
</tr>
</tbody>
</table>

5.18.8  Event source IRIS_SEMIHOSTING_INPUT_REQUEST

This event is issued whenever the semihosting implementation starts blocking on a blocking read operation or returns no data for a non-blocking read operation. It indicates that an application requests some data.

Table 5-11  Event source IRIS_SEMIHOSTING_INPUT_REQUEST

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDES</td>
<td>NumberU64</td>
<td>File descriptor number used for input. This is usually 0, meaning stdin. This value should be passed to the semihosting_provideInputData() function when passing data.</td>
</tr>
<tr>
<td>NONBLOCK</td>
<td>Boolean</td>
<td>Optional. If present and True, this request is caused by a non-blocking read operation. Otherwise, the target instance remains in a blocking read until it receives enough data.</td>
</tr>
<tr>
<td>RAW</td>
<td>Boolean</td>
<td>Optional. If True, all subsequent user input should be fed into the model immediately when it becomes available, as if a terminal is switched to raw I/O. When this field is missing, or False, user input should be fed into the model when the user presses the Enter key, in other words terminal cooked mode. The user interface can allow editing the line before sending it to the model. The raw or cooked mode should stay active until the next IRIS_SEMIHOSTING_INPUT_REQUEST or IRIS_SEMIHOSTING_INPUT_UNBLOCKED event for this file descriptor.</td>
</tr>
<tr>
<td>SIZEHINT</td>
<td>NumberU64</td>
<td>Number of bytes that the model consumes immediately.</td>
</tr>
</tbody>
</table>

5.18.9  Event source IRIS_SEMIHOSTING_INPUT_UNBLOCKED

This event is issued whenever a blocking semihosting read operation is unblocked.

It does not mean that no more input is required in the future. It is only a hint that the target instance is not currently blocked because of missing input. Non-blocking read operations never send this event because they never block. The raw or cooked mode should be set to cooked, see the RAW field of IRIS_SEMIHOSTING_INPUT_REQUEST.

Table 5-12  Event source IRIS_SEMIHOSTING_INPUT_UNBLOCKED

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDES</td>
<td>NumberU64</td>
<td>File descriptor number used for input. This is usually 0, which means stdin.</td>
</tr>
</tbody>
</table>

5.18.10  Event source IRIS_SEMIHOSTING_CALL

This event is issued just before a semihosting function is called. It can be used to monitor semihosting call usage.
Activating it does not affect the model behavior. It cannot be used to re-implement the built-in semihosting calls or to extend semihosting.

### Table 5-13 Event source IRIS_SEMIHOSTING_CALL

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERATION</td>
<td>NumberU64</td>
<td>The operation number of the built-in or user semihosting call according to the semihosting specification. This is the value of the operation number register when the target issues the semihosting trap instruction.</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>NumberU64</td>
<td>Register argument for the called function. This is the value of the parameter register when the target issues the semihosting trap instruction. This either contains an argument value for the called function, or it contains the virtual address of a data structure. This decision and the semantics depend on OPERATION. In case of a memory address, the semihosting implementation must read the memory using the memory_read() function.</td>
</tr>
</tbody>
</table>

#### 5.18.11 Event source IRIS_SEMIHOSTING_CALL_EXTENSION

This event source is intrusive. Activating it changes the model behavior. When it is active, this event is issued instead of a built-in semihosting function call.

The receiver of this event is expected to implement and execute the semihosting call from within the event callback. This event source should be activated with eventStream_create(syncEc=True).

---

**Note**

All synchronous event activity across IPC has a severe performance impact.

---

The ec_FOO() callback must do one of the following:

- Execute the semihosting function, and therefore override the built-in implementation. In this case it must call semihosting_return(). semihosting_return() must not be called from any other event callback, otherwise E_invalid_context is returned. Functions that need to return more data than an integer should write it directly into memory using memory_write(). A buffer consisting of a memory address and length is usually passed to the called function as an argument for this purpose.
- Leave execution to the built-in implementation, and therefore do not override it. In this case, it must call semihosting_notImplemented().
- Leave execution to the next client that registered this event source. If multiple clients have registered this event source, any of them might be called first and have the choice to implement the function or pass it to the next one, in no defined order. This enables different plug-ins to override separate, non-overlapping, sets of functions.

### Table 5-14 Event source IRIS_SEMIHOSTING_CALL_EXTENSION

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERATION</td>
<td>NumberU64</td>
<td>The operation number of the built-in or user semihosting call according to the semihosting specification. This is the value of the operation number register when the target issues the semihosting trap instruction.</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>NumberU64</td>
<td>Register argument for the called function. This is the value of the parameter register when the target issues the semihosting trap instruction. This either contains an argument value for the called function, or it contains the virtual address of a data structure. This decision and the semantics depend on OPERATION. In case of a memory address, the semihosting implementation must read the memory using the memory_read() function.</td>
</tr>
</tbody>
</table>
5.19 Simulation accuracy (sync levels) API

Some target instances support adjusting the trade-off between simulation speed and accuracy, by using a synchronization level or sync level.

The syncLevel_request() and syncLevel_release() functions request and release a minimum sync level. Calling syncLevel_request(N) ensures that the effective sync level is at least N. syncLevel_release(N) must be called for every call to syncLevel_request(N) when the requested sync level is no longer required. This might cause the sync level to decrease, and therefore increase the simulation speed, depending on the sync levels requested by other users.

Requesting and releasing a sync level of zero is valid but has no effect. The sync level of a particular target instance is a shared resource. It is not possible to set a specific sync level, except the highest one, because another client might be using a higher sync level.

Target instances that do not support requesting sync levels must return E_function_not_supported_by_instance for the syncLevel_*() functions.

This section contains the following subsections:

- 5.19.1 Sync points
- 5.19.2 syncLevel_get()
- 5.19.3 syncLevel_release()
- 5.19.4 syncLevel_request()
- 5.19.5 SyncLevelState

5.19.1 Sync points

A sync point is a point at which the simulation can detect whether it needs to stop and at which it can start and stop producing trace and events. The sync level determines where the sync points are in terms of simulated time.

### Table 5-15 Sync levels and sync points

<table>
<thead>
<tr>
<th>Sync level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OFF. Maximum simulation speed. No accuracy guarantees. In particular, the simulation cannot be stopped immediately, even from synchronous callbacks or model code, and the values of the PC and instruction count registers are generally out of date, even when read from synchronous callbacks or model code. <strong>Use cases:</strong>&lt;br&gt;• Simulations that do not require immediate stopping of any kind.&lt;br&gt;• Free-running simulations that are used for software development.&lt;br&gt;• Normal debugging sessions when no watchpoint is set. <strong>The sync point is the end of the current quantum.</strong></td>
</tr>
<tr>
<td>1</td>
<td>SYNC_STATE. Slightly slower than 0. It is possible to read up-to-date resource values from synchronous callbacks (syncEc=True) from the model and model code, for example the PC register. The simulation cannot be stopped immediately from within synchronous callbacks or model code. <strong>Use cases:</strong>&lt;br&gt;• External breakpoints that block the simulation.&lt;br&gt;• Inspecting the processor state from within peripheral accesses.</td>
</tr>
</tbody>
</table>
### Table 5-15 Sync levels and sync points (continued)

<table>
<thead>
<tr>
<th>Sync level</th>
<th>Description</th>
</tr>
</thead>
</table>
| 2          | POSTInsn_IO. Similar to 1 but slightly slower, and the simulation can be stopped immediately while executing I/O (LD/ST) instructions from within synchronous callbacks and model code by using the `simulationTime_stop()` function. The simulation stops after the currently executed I/O (LD/ST) instruction has completed. Use cases:  
  - Watchpoints.  
  - External breakpoints, in other words, breakpoints that are set in peripherals, behind a bridge.  
  - Complex breakpoints built from LD/ST-related events.  
  The sync point is the same as for sync level 1, but additionally after every I/O instruction. |
| 3          | POSTInsn_ALL. Similar to 2 but slightly slower, and the simulation can be stopped immediately, independently of the instruction being executed. The simulation stops after the currently executed instruction has completed. Use case:  
  - Complex breakpoints that are built from arbitrary events.  
  The sync point is the same as for sync level 2, but additionally after every instruction. |

Some use cases that require specific sync levels:
  - Watchpoints, or memory breakpoints, require sync level 2. This is handled transparently by the model and the sync level does not need to be requested by the watchpoint setter.
  - External breakpoints that can stop the simulation use sync level 2.
  - Trace event-based breakpoints usually require sync level 2 or 3.
  - Events that inspect the state of cores, in particular the PC and instruction count, from within the `ec_FOO()` callback require sync level 1.

**Related references**
- `5.11.4 simulationTime_stop()` on page 5-124
- `5.17.5 ec_FOO()` on page 5-148

### 5.19.2 syncLevel_get()

Queries the current sync-level state. This is only provided for debugging purposes. Clients must not rely on the values returned by this function because other clients might explicitly change the sync level, or might implicitly change it using watchpoints.

**Arguments**

```plaintext
instId
```
Type: `NumberU64`
Opaque number uniquely identifying the target instance.

**Return value**

```plaintext
SyncLevelState
```
Current sync-level value and the number of registered users for each sync level. See `SyncLevelState` on page 5-171.

**Errors**

- `E_unknown_instance_id`
- `E_unknown_sync_level`
5.19.3 syncLevel_release()

Releases a previous request for a minimum sync level. This signals that the client no longer requires that the sync level of the instance goes no lower than the released level.

Arguments

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

syncLevel
Type: NumberU64
Requested minimum sync level:

"0" ("OFF")
Maximum simulation speed. No accuracy guarantees. In particular, the simulation cannot be stopped immediately, even from synchronous callbacks or model code, and the values of the PC and instruction count registers are generally out of date, even when read from synchronous callbacks or model code.

Use cases: Simulations that do not require immediate stopping of any kind. Freerunning simulations used for software development. Normal debugging sessions when no watchpoint is set.

The sync point, which is the point at which the simulation can detect that it needs to stop and can start and stop generating trace and events, is the end of the current quantum.

"1" ("SYNC_STATE")
Slightly slower than 0. It is possible to read up-to-date resource values from synchronous callbacks from the model (syncEc=True) and model code, for example the PC register. The simulation cannot be stopped immediately from within synchronous callbacks or model code.

Use cases: External breakpoints that block the simulation. Inspecting the processor state from within peripheral accesses.

Sync point is the same as for sync level 0.

"2" ("POST_INSN_IO")
Slightly slower than 1. Like 1 but the simulation can be stopped immediately while executing I/O (LD/ST) instructions from within synchronous callbacks and model code, using the simulationTime_stop() on page 5-124 function. The simulation stops after the currently executing I/O (LD/ST) instruction has completed.

Use cases: Watchpoints. External breakpoints (breakpoints in peripherals). Complex breakpoints built from LD/ST-related events.

Sync point is the same as for sync level 1 plus after every I/O instruction.

"3" ("POST_INSN_ALL")
Slightly slower than 2. Like 2 but the simulation can be stopped immediately, independently of the instruction currently executing. The simulation stops after the currently executing instruction has completed.

Use cases: Complex breakpoints built from arbitrary events.

Sync point is the same as for sync level 2 plus after every instruction.
Return value

Function has no return value.

Errors

- E_unknown_instance_id.
- E_unknown_sync_level.

5.19.4 syncLevel_request()

Requests a minimum sync level. This signals that the sync level of the instance must go no lower than the level requested.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

syncLevel

Type: NumberU64

Requested minimum sync level:

"0" ("OFF")

Maximum simulation speed. No accuracy guarantees. In particular, the simulation cannot be stopped immediately, even from synchronous callbacks or model code, and the values of the PC and instruction count registers are generally out of date, even when read from synchronous callbacks or model code.

Use cases: Simulations that do not require immediate stopping of any kind. Free-running simulations used for software development. Normal debugging sessions when no watchpoint is set.

The sync point, which is the point at which the simulation can detect that it needs to stop and can start and stop generating trace and events, is the end of the current quantum.

"1" ("SYNC_STATE")

Slightly slower than 0. It is possible to read up-to-date resource values from synchronous callbacks from the model (syncEc=True) and model code, for example the PC register. The simulation cannot be stopped immediately from within synchronous callbacks or model code.

Use cases: External breakpoints that block the simulation. Inspecting the processor state from within peripheral accesses.

Sync point is the same as for sync level 0.

"2" ("POST_INSN_IO")

Slightly slower than 1. Like 1 but the simulation can be stopped immediately while executing I/O (LD/ST) instructions from within synchronous callbacks and model code, using the simulationTime_stop() on page 5-124 function. The simulation stops after the currently executing I/O (LD/ST) instruction has completed.

Use cases: Watchpoints. External breakpoints (breakpoints in peripherals). Complex breakpoints built from LD/ST-related events.

Sync point is the same as for sync level 1 plus after every I/O instruction.
"3" ("POST_INSN_ALL")

Slightly slower than 2. Like 2 but the simulation can be stopped immediately, independently of the instruction currently executing. The simulation stops after the currently executing instruction has completed.

Use cases: Complex breakpoints built from arbitrary events.
Sync point is the same as for sync level 2 plus after every instruction.

**Return value**
Function has no return value.

**Errors**
- E_unknown_instance_id.
- E_unknown_sync_level.

### 5.19.5 SyncLevelState

SyncLevelState members:

- **counts**
  Type: NumberU64[]
  Current number of requests for sync levels 0, 1, 2, and 3 respectively. The count for sync level 0 is always 0. Array length is always four.

- **syncLevel**
  Type: NumberU64
  Current effective sync level. This is the highest requested sync level.
5 Iris APIs

5.20 Instance registry, instance discovery, and interface discovery API

All entities in Iris are represented by instances. Instances must first be registered in the global instance registry. Instances can discover and communicate with other instances, and can check which functions they support.

This section contains the following subsections:

- 5.20.1 Hierarchical instance names and instance classes on page 5-172.
- 5.20.2 Registering instances on page 5-173.
- 5.20.3 Unregistering instances on page 5-174.
- 5.20.4 Instance properties on page 5-175.
- 5.20.5 Interface discovery on page 5-176.
- 5.20.6 Use cases for instance_ping() on page 5-176.
- 5.20.7 Interface versioning on page 5-177.
- 5.20.8 Naming conventions for new functions on page 5-177.
- 5.20.9 instanceRegistry_getInstanceInfoByInstId() on page 5-178.
- 5.20.10 instanceRegistry_getInstanceInfoByName() on page 5-178.
- 5.20.11 instanceRegistry_getList() on page 5-179.
- 5.20.12 instanceRegistry_registerInstance() on page 5-179.
- 5.20.13 instanceRegistry_unregisterInstance() on page 5-180.
- 5.20.14 instance_checkFunctionSupport() on page 5-180.
- 5.20.15 instance_getCppInterfaceIrisInstance() on page 5-181.
- 5.20.16 instance_getFunctionInfo() on page 5-181.
- 5.20.17 instance_getProperties() on page 5-182.
- 5.20.18 instance_ping() on page 5-182.
- 5.20.19 instance_registerProxyEventSource() on page 5-183.
- 5.20.20 instance_registerProxyFunction() on page 5-183.
- 5.20.21 instance_unregisterProxyEventSource() on page 5-184.
- 5.20.22 instance_unregisterProxyFunction() on page 5-184.
- 5.20.23 AttributeInfo on page 5-185.
- 5.20.24 CppInterfacePointer on page 5-186.
- 5.20.25 EnumElementInfo on page 5-186.
- 5.20.26 FunctionInfo on page 5-187.
- 5.20.27 FunctionSupportRequest on page 5-187.
- 5.20.28 InstanceInfo on page 5-187.
- 5.20.29 Event source IRIS_INSTANCE_REGISTRY_CHANGED on page 5-188.

5.20.1 Hierarchical instance names and instance classes

Every entity in a simulation is uniquely identified by a hierarchical instance name string. Hierarchy levels are separated by dots.

The first hierarchy level in the string defines the instance class that this instance belongs to. It can be one of the following:

component

An entity that primarily is controlled and observed, typically:

- Components that are part of the component tree of the system and that model a specific piece of hardware, for example a core, memory, or peripheral.
- In-process plug-ins or out-of-process clients that model a component.

Following component. is the name of the top-level component of the component tree, or root if it has no name. This is followed by the instance names of all components in the hierarchical path up to and including the instance name that is being registered, for example component.cluster0.cpu0.
client
An entity that primarily observes other instances or controls the simulation. Typically:

• Debuggers.
• Trace consumers.
• In-process plug-ins that primarily observe and control. In other words, DSOs that are loaded using the FM_PLUGINS environment variable, or built-in plug-ins that are statically linked into the simulator executable.
• Out-of-process clients, in other words, anything that connects to the IrisTcpServer.

Following client is the instance name of the client, which might or might not be hierarchical, for example client.plugin.ListInstances.

framework
An entity that is part of the simulation framework. The following instance names are defined for this instance class:

• framework.GlobalInstance.
• framework.SimulationEngine.

--- Note ---

• All instances can discover and communicate with all other instances. The instance class only gives an indication of the role of the instance to other instances. It does not limit what the instance can do, for example only producing or only consuming events.
• Physical entities, for example DSO plug-ins, can register multiple instances, for example one or more components, and also a client, if the entity contains one.

5.20.2 Registering instances

The global function instanceRegistry_registerInstance() registers a new instance in the global instance registry and assigns an instance id, instId, to it.

A call to this function to register instance foo.bar, has the following effects:

• The instance is registered under its path name foo.bar and a unique instId is assigned to it, which the function returns. The list of all registered instances can be queried using instanceRegistry_getList().
• All framework instances can use the instId to route requests and responses to foo.bar.

instanceRegistry_registerInstance() must be called as a request, not as a notification. When it is called as a notification, the call is ignored and no instance is registered. Instances must call it before calling any other Iris function because their instance id must be included in all request ids. When calling it, instances do not yet have an instance id assigned, so they cannot set their instance id in bits[63:32] of the request id. Instead, they set bits[63:32] of the request id to zero. The caller can freely choose bits[31:0], as for normal requests.

--- Note ---

The IrisSupportLib library normally takes care of constructing request objects for you, so unless you are implementing your own library, you do not need to create request ids yourself.

--- ---

The following restrictions apply to registering instances:

• The top-level in the hierarchical instance name must be one of the instance classes, for example component.
• Instances must register themselves as early as possible, in other words as soon as they exist and an IrisInterface can send the instanceRegistry_registerInstance() call.

See 5.20.3 Unregistering instances on page 5-174 for restrictions on unregistering instances.
The hierarchy does not indicate dependencies. Children do not depend on the presence of their parents. So, the following actions are valid:

- Registering `component.foo.bar` without having registered `component.foo` beforehand.
- Unregistering `component.foo` before unregistering `component.foo.bar`, assuming both have previously been registered.

### 5.20.3 Unregistering instances

`instanceRegistry_unregisterInstance()` unregisters an instance from the instance registry. It undoes all the effects of `instanceRegistry_registerInstance()`.

When `instanceRegistry_unregisterInstance()` returns, the instance is no longer visible to `instanceRegistry_getList()`. Functions that receive an `instId` argument fail with `E_unknown_instance_id` if they are called with an unregistered `instId`.

Calling `instanceRegistry_unregisterInstance(instId)` and waiting for its response is the only way to make sure the `IrisInterface` of the instance with `instId` is no longer in use.

Side effects of unregistering an instance:

- Unregistering instance `X` using `instanceRegistry_unregisterInstance()` automatically destroys all event streams that are sent to instance `X`. That is, event streams that were created with `eventStream_create(ecInstId=X)`.
- Unregistering an instance does not affect any breakpoints that the instance has set.

The following constraints apply when unregistering instances:

- Instances must unregister themselves as late as possible, in other words just before or during destruction, as long as an `IrisInterface` can send the `instanceRegistry_unregisterInstance()` function call.
- Constraints on communication during unregistering:
  - Before sending the `instanceRegistry_unregisterInstance()` request, the instance can call functions normally, and it must accept function call responses and respond to function calls normally.
  - After sending the `instanceRegistry_unregisterInstance()` request and before receiving a response to the request, the instance must not call any functions. It must accept function call responses and respond to function calls normally.
  - After receiving the response to the `instanceRegistry_unregisterInstance()` request, the instance must not call any functions. It must assume that `irisHandleMessage()` on its `IrisInterface` interface will not be called after this point. It can destroy itself and the `IrisInterface` after this point.
- Instances that receive a function call for an unregistered `instId` must return `E_unknown_instance_id`.
- At any time, other instances can call functions for a specific `instId`. They either receive an OK response, or an error response from the destination instance or from a message-routing instance, for example the GlobalInstance. If the instance no longer exists, they receive an `E_unknown_instance_id` error. This is normal behavior and must not cause any side effects, for example closing the simulation. The caller is responsible for handling this error in a suitable way.

If an instance fails to call `instanceRegistry_unregisterInstance(instId)` before destroying itself:

- If the instance is in-process, that is, connected using the C++ `IrisInterface` interface, this is considered to be as severe a programming error as using a freed C++ object, and is explicitly forbidden.
- If the instance is out-of-process, that is, connected using a TCP socket or any other mechanism, this is considered normal behavior. For example, this might happen if the remote process crashes or the TCP connection closes from the remote side. The message-routing instance that provided the connection is able to detect such a loss of connection. It must then:
  - Synthesize a call to `instanceRegistry_unregisterInstance()` into the rest of the system.
  - Send `E_unknown_instance_id` error responses to all pending function calls that it is handling for the instance that was destroyed.
Related references
5.17.7 eventStream_create() on page 5-150

5.20.4 Instance properties

The function instance_getProperties() gets detailed information that is inherent to the instance and does not change, for example the type of component and whether certain features are supported or not. Properties should not be confused with parameters, which are variable characteristics of a component, set at compile time or runtime.

instance_getProperties() returns a set of arbitrary key/value pairs. The following tables list all properties that have defined semantics in Iris. Instances can report additional properties that are not listed here. All instance properties are optional.

Table 5-16 Instance properties defined by Iris, typically only for component instances

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>register.canonicalRnScheme</td>
<td>String</td>
<td>Canonical register number scheme used by the canonicalRn member of RegisterInfo. Canonical register numbers are intended to be target-specific numbers that identify registers in the device. 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. Arm components use arm.com/registers, if they expose registers.</td>
</tr>
<tr>
<td>memory.canonicalMsnScheme</td>
<td>String</td>
<td>Canonical memory space number scheme used by the canonicalMsn member of MemorySpaceInfo. Canonical memory space numbers are intended to be target-specific numbers that identify memory spaces in the device. 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. Arm components use arm.com/memoriespaces, if they expose memory spaces, see 5.7.6 Canonical memory space number scheme on page 5-89.</td>
</tr>
</tbody>
</table>

Table 5-17 Instance properties defined by LISA+, typically only for component instances

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>componentName</td>
<td>String</td>
<td>The name of the component that this is an instance of. For example RAMDevice.</td>
</tr>
<tr>
<td>version</td>
<td>String</td>
<td>Component version, as defined in the LISA+ properties section.</td>
</tr>
<tr>
<td>componentType</td>
<td>String</td>
<td>Component type. For example:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Bridge</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Clocking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Media</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Peripheral</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Signals</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SystemIP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Components can report other component types. If a strong classification is needed for these, they should be included in the same class as Other. The LISA+ name is component_type.</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>Short description of the functionality of the component. Can contain linefeeds, but is usually just a single line of text.</td>
</tr>
</tbody>
</table>
Table 5-17  Instance properties defined by LISA+, typically only for component instances (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>documentationFile</td>
<td>String</td>
<td>Filename of the documentation for this component. A user interface can open this file upon request. The LISA+ name is documentation_file.</td>
</tr>
<tr>
<td>executesSoftware</td>
<td>NumberU64</td>
<td>A hint that is set to 1 if the component can execute software. Clients can take this as a hint that this component is a CPU-like debug target, in contrast to a peripheral, which might also be inspected but which generally does not execute software. The LISA+ name is executes_software.</td>
</tr>
<tr>
<td>loadfileExtension</td>
<td>String</td>
<td>A hint for clients about which filename extensions are usually suitable for the image_loadFile() function. Clients should offer them to users in addition to All Files. Clients that use their own loader and always send binary data to the target should ignore this property. The list contains glob-style wildcard expressions, separated by semicolons, for example &quot;<em>.txt&quot; or &quot;</em>.axf;*.elf&quot;. The LISA+ name is loadfile_extension.</td>
</tr>
</tbody>
</table>

Table 5-18  Instance properties for client instances and for instances connected using IPC

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Human readable name of a client. Single line, usually prettier than the instance name, for example &quot;Arm Development Studio v1.0&quot;. This does not need to be unique for different clients, if they are of the same type.</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>Multiple-line description of this client.</td>
</tr>
<tr>
<td>connectionInfo</td>
<td>String</td>
<td>Single line that describes the way this instance is connected to the simulation. This can be either: &quot;in-process&quot; For DSO plug-ins, for example. This should be the default interpretation when this property is missing. &quot;iris://&lt;ip&gt;:&lt;port&gt;&quot; For an IPC client on &lt;ip&gt; and &lt;port&gt;.</td>
</tr>
</tbody>
</table>

Related references
5.6.16 RegisterInfo on page 5-78
5.7.20 MemorySpaceInfo on page 5-100
5.10.9 image_loadFile() on page 5-120

5.20.5  Interface discovery

Instances can support any subset of Iris functions. It is possible to check whether an instance supports a specific function or set of functions. It is also possible to get a list of all functions that a specific instance supports.

It is not mandatory to check whether a function is supported before calling it. Functions that are not supported by an instance return E_function_not_supported_by_instance. Interface discovery is mandatory for all instances. In other words, instance_checkFunctionSupport() and instance_getFunctionInfo() must not return E_function_not_supported_by_instance.

5.20.6  Use cases for instance_ping()

This function has no effect on the instance. It is intended to be a no-op.
It has the following use cases:

- A dummy operation to simulate keep-alive. To keep the TCP connection open without disturbing the other side of the connection, this function can be called at regular intervals, for example every 7200s. However, it is preferable to use the TCP keep-alive socket options, if available, instead. The ping should be addressed at an instance on the other side, for example the IrisTcpServer instance from a client or the client instance from the IrisTcpServer.

- Benchmarking. Ping can optionally return a dummy payload, which could, for example, be any or all of the following:
  - Sequence count.
  - Timestamp.
  - Dummy data that is used by a benchmark to measure the transport performance.

Instances that do not support `instance_ping()` must return `E_function_not_supported_by_instance`, although instances are encouraged to implement this trivial function.

### 5.20.7 Interface versioning

Interface versioning is implicit and per-function in the Iris interfaces. Explicit interface versioning is not necessary for Iris interfaces and therefore is not provided.

Versioning works as follows:

- Use `instance_checkFunctionSupport()` to determine whether a specific function or set of functions is supported. You can also use `instance_getFunctionInfo()` to check which mandatory and optional arguments are supported by a specific function.
- Callers only take the information they need from return value objects. They must ignore all object members they do not know about.
- New functionality might be added to Iris without breaking existing clients in the following ways:
  - If new details are returned, for example static register information, the return value objects are extended with new members. Existing clients will ignore these additions and new clients can reliably see them.
  - If new optional features are added to a function call, they are added as optional arguments, with a default value that is fully compatible with the previous behavior of the function. Existing clients will not specify these new arguments and will receive the previous behavior.
  - If the semantics of a function change so significantly that it cannot remain compatible with the previous behavior by adding optional arguments, a new function with a new name must be added. Existing clients will not know about this new function and will call the existing function. If the existing function is no longer supported, they will find out reliably.
  - New orthogonal functions are added, extending the target without breaking existing clients.

For a new feature to become useful and usable, it must be supported by both the client and the target.

### 5.20.8 Naming conventions for new functions

When adding new functions that replace or extend existing functions, use the following naming conventions.

---

**Note**

- Only add new functions if the same effect cannot be achieved by extending an existing function with new return value members or with new optional arguments.
- All orthogonal changes or additions can be achieved with new optional arguments. Adding new functions to replace or enhance existing ones is not necessary.

---

Assuming the existing function is called `breakpoint_set()`, and a new non-orthogonal feature, `foo`, must be added, which cannot be supported by extending `breakpoint_set()` with optional arguments, for example `breakpoint_set(enableFoo=True, ...)`, new versions of this function should be named, in the following order, from most to least preferred:
breakpoint_setFoo()
Use this when Foo is a suitable description of the new non-orthogonal feature. If the feature is orthogonal to existing functionality, an argument foo={"space":42, "index":6} or enableFoo=True should be added to the existing breakpoint_set() function, rather than introducing a new function.

breakpoint_set_armCortexA53()
Use this when the behavior is processor-specific, and cannot be generalized. Generalizing is preferred if possible, preferably in an orthogonal way, so that introducing a new function is not necessary. For example:

```
breakpoint_set(addressArmCortexA53UtlbTag=<something_very_complex>)
```

experimental_breakpoint_set_armCortexA53()
Use this when the function is experimental and should only be used by an experimental client.

breakpoint_foo()
Use this when the new functionality is breakpoint-related, but does not have set semantics but foo semantics instead.

foo_bar()
Use this when this is entirely new functionality, not related to breakpoints at all.

breakpoint_set2()
Use this when this function has the same semantics as breakpoint_set, but has a better interface with incompatible arguments or incompatible return value and therefore enhancing the argument list or return value is not an option.

See also 5.2 Naming conventions on page 5-61 for general naming conventions.

5.20.9 instanceRegistry_getInstanceInfoByInstId()
Gets the instance name for an instance id.

**Arguments**

aInstId
Type: NumberU64
Instance id of the instance to look up. This argument is intentionally not called instId because this function is not targeted at the instance identified by this argument, but is targeted at the instance registry.

**Return value**

InstanceInfo

*InstanceInfo on page 5-187* object containing the name and instId of an instance.

**Errors**

- E_unknown_instance_id.

5.20.10 instanceRegistry_getInstanceInfoByName()
Gets the instance Id for a named instance.

**Arguments**

instName
Type: String
Instance name of the instance to look up.
Return value
InstanceInfo

*InstanceInfo on page 5-187* object containing the name and `instId` of an instance.

Errors
- `E_unknown_instance_name`

5.20.11 instanceRegistry_getList()

Gets a list of all instances, optionally filtered by a specific prefix.

Arguments
prefix

Type: `String`

Optional. Only return instances whose hierarchical instance name starts exactly with `prefix` or is exactly `prefix`. Only complete parts of the hierarchical names are matched, that is, `prefix="component.root.ram"` does not match "component.root.ramdevice", but does match "component.root.ram.bank0" and also matches "component.root.ram".

The primary use case of `prefix` is to conveniently query only a specific class of instances, for example all components of a simulation using `prefix="component"`.

If this is missing or empty, all instances are returned. If no matching instance is found, an empty list is returned, rather than an error.

Return value
InstanceInfo[]

*InstanceInfo on page 5-187* Objects describing the instances matching `prefix`.

5.20.12 instanceRegistry_registerInstance()

Registers a new instance in the global instance registry and assigns an instance id (`instId`) to it.

Arguments

instName

Type: `String`

Hierarchical instance name. Hierarchy levels are separated by dots (`'.'`). The first hierarchy level defines the class this instance belongs to, namely `component`, `client`, or `framework`. Instance names must not start or end with a dot and must not contain two adjacent dots. The instance name elements that are separated by dots must be C identifiers. For invalid instance names, `E_invalid_instName` is returned.

channelId

Type: `NumberU64`

Optional. For in-process instances, this is mandatory. It indicates the IrisInterface communication of the caller of this function. For out-of-process instances, for example ones that are connected using TCP, this is ignored and must be omitted.
uniquify
Type: Boolean
Default: False
Optional. If True, uniquely name if instName already exists, by appending a suffix to instName such that instance + suffix is a unique instance name. This must only be used by client instances, for example debuggers and plug-ins, not by component instances, which must choose a unique name based on the system hierarchy. Instances must use a meaningful stem as instName, for example the name of a debugger or a plug-in instance name.

Return value
InstanceInfo
Object containing the instId and the instName of the registered instance. The opaque instance id uniquely and globally identifies the instance in the simulation until it is unregistered using instanceRegistry_unregisterInstance() on page 5-180. See InstanceInfo on page 5-187.

Errors
• E_invalid_instName.
• E_instance_already_registered.

5.20.13 instanceRegistry_unregisterInstance()
Unregisters an instance from the instance registry.

Arguments
aInstId
Type: NumberU64
Opaque number uniquely identifying the instance to be unregistered. This argument is intentionally not called instId because this function is not targeted at the instance identified by this argument, but is targeted at the instance registry.

Return value
Function has no return value.

Errors
• E_unknown_instance_id.

5.20.14 instance_checkFunctionSupport()
Checks whether a specific instance supports a specific set of functions and, optionally, function arguments. If a Boolean result is not sufficient, use instance_getFunctionInfo() to retrieve a list of all supported functions and their arguments. This function does not return errors for unknown functions or arguments listed in functions.

Arguments
functions
Type: FunctionSupportRequest[]
List of function names with optional argument list per function.

instId
Type: NumberU64
Opaque number uniquely identifying the instance.
Return value

Boolean

True: Instance supports all functions and all their arguments that are specified in functions. The instance might support functions not specified in functions and might also support more arguments than are specified in functions. False: Instance either does not support at least one of the specified functions or it does not support at least one of the specified function arguments.

Errors

• E_unknown_instance_id.

5.20.15 instance_getCppInterfaceIrisInstance()

Get a host pointer to the C++ iris::IrisInstance implementing an instance.

This is an exotic function which must generally not be used. This function is only supported by instances that use the C++ IrisSupportLib class iris::IrisInstance. If this function is present it always returns a non-null pointer. The caller of this function must make sure that the caller and callee use the same C++ interface class layout and run in the same process. This effectively means that they both must be compiled using the same compiler using the same header files. The returned pointer is only meaningful if caller and callee run in the same process. The meta-information provided alongside the returned pointer in CppInterfacePointer can (and must) be used to do minimal compatibility checking between caller and callee. Notes: - "Instances do not have to be implemented using 'iris"::IrisInstance', thus not all instances will support this function. - This function must only be used by instances that are tightly coupled because the use of this function creates a binary dependency between the instances. - Using this function is very exotic and must be avoided if possible. - Normal Iris function calls must always be preferred over direct C++ communication between instances.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the target instance.

Return value

CppInterfacePointer

Pointer to the C++ iris::IrisInstance (and associated meta-information) of this instance. See CppInterfacePointer on page 5-186.

Errors

• E_unknown_instance_id.

5.20.16 instance_getFunctionInfo()

Returns a list of all functions supported by a specific instance and their meta information, for example the description, the mandatory and optional arguments, and the type of the return value.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the instance to query. Specifying the value 0 queries the list of functions supported by the global instance, for example instanceRegistry_registerInstance() on page 5-179.
prefix
  Type: String
  Optional. Get FunctionInfo only for functions whose name starts with prefix. If this argument
  is missing or empty, all supported functions are returned.

Return value
Map[String]FunctionInfo
Object that maps function names onto FunctionInfo on page 5-187 Objects. The map might be empty,
for example if prefix does not match any function.

Errors
  • E_unknown_instance_id.

5.20.17 instance_getProperties()
Gets detailed information about an instance. This information is inherent to the instance and does not
change, for example the type of a component or whether certain features are supported or not.

Arguments
instId
  Type: NumberU64
  Opaque number uniquely identifying the instance.

Return value
Map[String]Value
Object containing all properties and their values. Numeric values can be stored as a number or as a
String in this object.

Errors
  • E_unknown_instance_id.

5.20.18 instance_ping()
This function has no effect on the instance. It is intended to be a no-op.

Arguments
instId
  Type: NumberU64
  Opaque number uniquely identifying the target instance.
payload
  Type: Value
  Optional. If present, this value is returned. If not present, Null is returned.

Return value
Value
If payload is present, it is returned, else Null is returned.
5.20.19 instance_registerProxyEventSource()

Register a proxy event source.

The event source will be transparently exposed by the proxy instance (instId) while the event source will be implemented (and also exposed) by the target instance (the instance actually generating the events). The main use-case is to add global event sources to the global instance while implementing them in other instances. This function is an internal infrastructure function. Clients never need to call this function. Calling this function for an already registered event source (proxy or normal event source) will return E_event_source_already_registered. The target instance must already be registered in the global instance and the event source must already be registered in the target instance before calling this function. The actual event callback calls are called by the target instance directly on the event receivers, not via the proxy instance.

Arguments

instId

Type: NumberU64

Opaque number uniquely identifying the proxy instance.

name

Type: String

Name of the proxy event source. If this event source is not supported by the target instance E_unknown_event_source is returned.

targetInstId

Type: NumberU64

Optional. Instance id of the instance which exposes the original event. If this is not a valid instance id E_unknown_instance_id is returned. If this is missing, the instance id in the request id is used as target instance.

Return value

Function has no return value.

Errors

• E_unknown_instance_id.
• E_unknown_event_source.
• E_event_source_already_registered.

5.20.20 instance_registerProxyFunction()

Register a proxy function which transparently forwards all calls to a target instance.

This can be used to expose a function on a proxy instance while it is implemented in the target instance. The main use-case is to add global functions to the global instance and implement them in another instance. This function is an internal infrastructure function. Clients never need to call this function. Calling this function for an already registered function (proxy or normal function) returns E_function_already_registered. The target instance must already be registered in the global instance and the function must already be registered in the target instance before calling this function.
Arguments

**instId**
Type: NumberU64
Opaque number uniquely identifying the proxy instance.

**name**
Type: String
Name of the proxy function. If this function is not supported by the target instance, E_unknown_function_name is returned.

**targetInstId**
Type: NumberU64
Optional. Calls to the proxy function are transparently forwarded to this instance. If this is not a valid instance id, E_unknown_instance_id is returned. If this is missing, the instance id in the request id is used as the target instance.

Return value
Function has no return value.

Errors
• E_unknown_instance_id.
• E_unknown_function_name.
• E_function_already_registered.

5.20.21 instance_unregisterProxyEventSource()
Unregister proxy event source. This is the counterpart of instance_registerProxyEventSource().

Arguments

**instId**
Type: NumberU64
Opaque number uniquely identifying the proxy instance.

**name**
Type: String
Name of the proxy event source. If this event source was not registered as a proxy event source E_unknown_event_source is returned.

Return value
Function has no return value.

Errors
• E_unknown_instance_id.
• E_unknown_event_source.

5.20.22 instance_unregisterProxyFunction()
Unregister proxy function. This is the counterpart of instance_registerProxyFunction(). This function is an internal infrastructure function. Clients never need to call this function.
Arguments

instId
Type: NumberU64
Opaque number uniquely identifying the proxy instance.

name
Type: String
Name of the proxy function. If this function was not registered as a proxy function, E_unknown_function_name is returned.

Return value
Function has no return value.

Errors

• E_unknown_instance_id.
• E_unknown_function_name.

5.20.23 AttributeInfo

AttributeInfo members:

defval
Type: Value
Optional. Default value for optional attributes. If not specified, the default values are 0, false, empty string, empty object, empty array, or null for number, boolean, string, object, array, and value respectively. Must be missing for mandatory arguments.

description
Type: String
Optional. Free-form description of the semantics and the structure of the argument. The format is the same as for the description member in FunctionInfo.

enums
Type: EnumElementInfo[]
Optional. List of valid enum values if this attribute is an enum. The position of the EnumElementInfo has no semantics. EnumElementInfo on page 5-186 objects always contain the value they describe.

optional
Type: Boolean
Optional. If present and True, this attribute is optional. If False or missing, this attribute is mandatory.

type
Type: String
Specifies the type of the attribute. Can be one of: Number, NumberU64, NumberS64, String, Boolean, Object, Array, or Value, or the name of a specified object format. Arrays are indicated by suffixing the type with [] (that is an array of NumberU64 would be NumberU64[]) and map-like objects are indicated by Map[{key-type}]value-type (that is Map[String]NumberU64). The map key type is always String.
5.20.24 **CppInterfacePointer**

CppInterfacePointer members:

**pointer**

*Type: NumberU64*

Host pointer of the C++ interface. This is a pointer to a class that supports runtime-type-information (RTTI). The pointer value is represented as a NumberU64. The consumer of this pointer must use the meta-information provided in this object to do a minimal check that it is safe to use the pointer value in the current context.

--- **Note** ---

The checks associated with `typeidName`, `typeidHashCode` and `sizeof` are insufficient to guarantee that the producer and consumer have compatible class layouts but are better than not doing any check at all. The consumer must also use other means to ensure compatibility with the producer, for example by being built into the same executable or set of executables.

**typeidName**

*Type: String*

The value of the expression `typeid(*pointer).name()` associated with `pointer`. This must be used by the consumer to verify that the producer and the consumer of the pointer have been compiled against the same source code using the same compiler.

**typeidHashCode**

*Type: NumberU64*

The value of the expression `typeid(*pointer).hash_code()` associated with `pointer`. This must be used by the consumer to verify that the producer and the consumer of the pointer have been compiled against the same source code using the same compiler.

**sizeof**

*Type: NumberU64*

The value of the expression `sizeof(*pointer)` associated with `pointer`. This must be used by the consumer to verify that the producer and the consumer of the pointer have been compiled against the same source code using the same compiler.

**pid**

*Type: NumberU64*

Process id associated with `pointer`. This must be used by the consumer to verify that the producer and the consumer of the pointer run in the same process. The pointer is only valid when used within the same process.

5.20.25 **EnumElementInfo**

EnumElementInfo members:

**description**

*Type: String*

Optional. Free-form description of the semantics and the structure of the argument. The format is the same as for the `description` member in `FunctionInfo`.
symbol
Type: String
Optional. Symbolic value, usually a C identifier, associated with a numeric value. This is mandatory for numeric values. This must not be present for enums of type String where the string value is the symbol's value.

value
Type: Value
Value of the enum element. This is usually either a String, NumberU64, or NumberS64 value. The type must match the type of the attribute this enum is associated with.

5.20.26 FunctionInfo
FunctionInfo members:
args
Type: Map[String]AttributeInfo
Object mapping the names of all arguments onto AttributeInfo on page 5-185 objects. The AttributeInfo on page 5-185 objects specify details of each formal argument like type, description, and enum values.
description
Type: String
Free-form description of this function. Must start with a capital letter and end with a dot. Can contain multiple lines (separated by LF). If so, the first line is a summary and is separated from the detailed description by an empty line.
errors
Type: String[]
The names of error codes that this function might respond with. See String.
retval
Type: AttributeInfo
AttributeInfo on page 5-185 object describing the return value (type, description).

5.20.27 FunctionSupportRequest
FunctionSupportRequest members:
args
Type: String[]
Optional. List of formal argument names to check for. At least the specified argument must be supported. More arguments can be supported. The default is not to check for any arguments, which is the same as an empty list.
name
Type: String
Name of a function to check for.

5.20.28 InstanceInfo
InstanceInfo members:
instId
- Type: NumberU64
- Opaque id uniquely identifying the instance.

instName
- Type: String
- Hierarchical instance name of the instance.

### 5.20.29 Event source IRIS_INSTANCE_REGISTRY_CHANGED

This event is generated when the list of registered instances has changed, shortly after the change was applied to the instance registry.

This event source is only provided by `framework.GlobalInstance`, not by individual instances.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EVENT</td>
<td>String</td>
<td>The event that occurred. Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;added&quot; Added a new instance to the instance registry.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;removed&quot; Removed an instance from the instance registry.</td>
</tr>
<tr>
<td>INST_ID</td>
<td>NumberU64</td>
<td>Instance id of the instance that was added or removed.</td>
</tr>
<tr>
<td>INST_NAME</td>
<td>String</td>
<td>Instance name of the instance that was added or removed.</td>
</tr>
</tbody>
</table>
5.21 Simulation instantiation and discovery API

This section describes the functionality to instantiate simulations and discover running simulations.

There are two main use cases when using simulations:

• Connecting to a running simulation using IPC.
• Instantiating a new simulation, in-process.

Most functions described in this section have a different scope to other Iris functions. Most Iris functions assume a pre-existing communication channel between instances in a pre-existing simulation, and this communication channel, regardless of its transport, is specific to one instantiated simulation.

Instead, the functions in this section deal with situations where a simulation is about to be instantiated or where a communication channel to an existing simulation is about to be established. The available functionality strongly depends on whether a caller connects using IPC or instantiates a new simulation in-process.

This section contains the following subsections:

• 5.21.1 Connecting to a running simulation using IPC on page 5-189.
• 5.21.2 Instantiating a new simulation, in-process on page 5-189.
• 5.21.3 Instantiation parameters on page 5-190.
• 5.21.4 Setting instantiation parameter values on page 5-190.
• 5.21.5 simulation_getInstantiationParameterInfo() on page 5-191.
• 5.21.6 simulation_instantiate() on page 5-191.
• 5.21.7 simulation_requestShutdown() on page 5-192.
• 5.21.8 simulation_reset() on page 5-192.
• 5.21.9 simulation_setInstantiationParameterValue() on page 5-192.
• 5.21.10 simulation_waitForInstantiation() on page 5-193.
• 5.21.11 InstantiationError on page 5-193.
• 5.21.12 InstantiationParameterValue on page 5-194.
• 5.21.13 InstantiationResult on page 5-195.
• 5.21.14 SimulationTimeObject on page 5-195.
• 5.21.15 Event source IRIS_SIMULATION_SHUTDOWN_ENTER on page 5-195.
• 5.21.16 Event source IRIS_SIMULATION_SHUTDOWN_LEAVE on page 5-196.

5.21.1 Connecting to a running simulation using IPC

Communicating with a running simulation typically involves these basic steps:

1. Optionally get a list of all available running simulations.
2. Connect to a specific simulation.
3. Use Iris to inspect and manipulate the simulation and the instances in it.
4. Disconnect from the simulation. Optionally request simulation shutdown.

5.21.2 Instantiating a new simulation, in-process

A client that does not yet have an instantiated simulation to communicate with is called a pre-instantiation client.

Examples of pre-instantiation clients:

• main() function of a standalone, non-SystemC, simulator executable.
• DSO entry point of a standalone, non-SystemC, simulator DSO.

Instantiating a new simulation follows this sequence:

1. Optionally get instantiation parameter meta-information, for example name, type, and description, from the simulator.
2. Optionally get instantiation parameter values from the user.
3. Optionally set instantiation parameter values.
4. Instantiate a simulation with instantiation parameter values.
5. Use Iris to inspect and manipulate the simulation and the instances in it.
6. Optionally request simulation shutdown.

5.21.3 Instantiation parameters

The parameters that are exposed through simulation_getInstantiationParameterInfo() are called instantiation parameters.

Instantiation parameters are used to initialize the initialization-time and runtime parameters of each instance during instantiation. After the system is instantiated, the instantiation parameter values are exposed through the resource_*( ) functions of each instance.

Instantiation parameters are exposed as a flat list of parameters. The full instance path is prepended to the parameter name, to indicate the hierarchy of the not-yet-instantiated system:

<instance_path>.<parameter_name>

This parameter will later on be exposed by instance <instance_path> as parameter <parameter_name>.

There are subtle differences between instantiation parameters and the instance-specific initialization-time and runtime parameters:

• Instantiation parameters have a hierarchical name and are exposed through simulation_getInstantiationParameterInfo(). They can only be set by using simulation_setInstantiationParameterValues(). At instantiation, there is no functional difference between initialization-time and runtime parameters. The instantiation parameters are the sum of all initialization-time and runtime parameters, but with hierarchical names.

• Instantiation parameters are global and their association with a specific, future, instance is only implied through their hierarchical parameter name. Instantiation parameters use ResourceInfo metadata, the function simulation_getInstantiationParameterInfo(), which is similar to resource_getList(), and the function simulation_setInstantiationParameterValues(), which is similar to resource_write().

• The main difference between instantiation parameters and the per-instance initialization-time and runtime parameters is that instantiation parameters are only identified by their hierarchical name string and do not have a resource id (rscId) or instance id (instId).

• Initialization-time parameters can only be set before instantiation and cannot be changed later. Before instantiation, they are exposed globally as instantiation parameters, with a hierarchical parameter name, and after instantiation they are exposed as resources with parameterInfo.initOnly = True. That is, they are exposed only by the instance they belong to, with a non-hierarchical name.

• Runtime parameters can be set before instantiation and can also be changed later. Before instantiation, they are exposed globally as instantiation parameters, with a hierarchical parameter name, and after instantiation they are exposed as resources with parameterInfo.initOnly = False. That is, they are exposed only by the instance they belong to, with a non-hierarchical name.

Related references
5.6 Resources API on page 5-67
5.6.18 ResourceInfo on page 5-80

5.21.4 Setting instantiation parameter values

Pre-instantiation clients can optionally call simulation_setInstantiationParameterValues() to initialize all init-time and runtime parameters for all instances during the instantiation of the simulation.

This function modifies state that exists before the simulation is instantiated, it does not instantiate the simulation. It can be called multiple times. Each invocation can set all of the instantiation parameters or a subset of them. Any parameters that this function does not set keep their default values. Each invocation can overwrite parameter values that were set with previous invocations of simulation_setInstantiationParameterValues().

An implementation can defer any errors, for example E_unknown_parameter_name or E_type_mismatch, until instantiation. These deferred errors are then returned by simulation_instantiate().
5.21.5  simulation_getInstantiationParameterInfo()

Gets the meta information of all instantiation parameters of a not-yet-instantiated simulation.

Calling this function is optional. If a pre-instantiation client already knows which parameters exist, it can instantiate the simulation without first getting the meta information using this function, for example when reading parameter values from a config file. This function has no side effects.

Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

prefix

Type: String

Optional. Only return parameters whose hierarchical name starts exactly with prefix or is exactly prefix. Only complete parts of the hierarchical names are matched, that is, prefix="component.root.ram" does not match "component.root.ramdevice", but does match "component.root.ram.bank0" and also matches "component.root.ram". If this is missing or empty, all parameters are returned. If no matching parameter is found, an empty list is returned rather than an error.

Return value

ResourceInfo[]

List of meta information for instantiation parameters as an array of ResourceInfo on page 5-80 objects. The semantics of these ResourceInfo entries differ slightly from instance-specific ResourceInfo entries:

- The list of instantiation parameters is global and an association with a specific future instance is only implied by its hierarchical name.
- ResourceInfo.rscId on page 5-80 is missing. (Instantiation parameters are identified by their hierarchical name.)
- ResourceInfo.name on page 5-80 contains the hierarchical name of the instantiation parameter, with the format <instance_path>.<parameter_name>.
- ResourceInfo.cname on page 5-80 is missing. Instantiation parameters do not have valid C identifiers and are always identified by their name string.

5.21.6  simulation_instantiate()

Pre-instantiation clients use this function to instantiate the simulation. It implicitly uses the parameter state that was set up by simulation_setInstantiationParameterValues().

Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

Return value

InstantiationResult

List of errors and warnings that occurred during instantiation. This includes invalid parameter values and license checking errors. See InstantiationResult on page 5-195.
5.21.7 simulation_requestShutdown()

Requests that the simulation destroys itself cleanly. The shutdown might be delayed and is likely to happen after this function returns. Receiving a response to this function call does not mark any special event and does not mean the simulation has been shut down.

Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

Return value

Function has no return value.

5.21.8 simulation_reset()

Resets the simulation to exactly the same state it had after instantiation, see simulation_instantiate().

Some simulations might require simulation time to be stopped before they can be reset. These simulations respond with the error code E_simulation_running if simulation_reset() is called while simulation time is progressing. For simulations that can be reset while simulation time is progressing, the simulation is stopped automatically before resetting the system. Simulation time is not resumed after the reset is done. After simulation_reset() completes, the simulation is always stopped.

Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

allowPartialReset

Type: Boolean

Default: False

Optional. If present and true, perform a partial simulation reset for simulations that do not support a full reset. This might be because some components in a simulation do not support reset functionality. By setting allowPartialReset to true, a client acknowledges that it accepts the consequences of not resetting the whole simulation.

Return value

Function has no return value.

Errors

- E_full_reset_not_supported.
- E_simulation_running.

5.21.9 simulation_setInstantiationParameterValues()

Sets the instantiation parameter values. These values are used to initialize all init-time and run-time parameters of all instances during the instantiation of the simulation. This function modifies a state that exists before the simulation is instantiated. It does not instantiate the simulation.
Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

data

Type: InstantiationParameterValue[]

List of instantiation parameter values.

Return value

Function has no return value.

Errors

• E_unknown_parameter_name.
• E_type_mismatch.

5.21.10 simulation_waitForInstantiation()

This function can be used by clients that connect to a simulation to make sure that the simulation has been instantiated and all components have been initialized.

It can be called at any time during the simulation. If called before the simulation has been instantiated the response will only be sent after the simulation has been instantiated. If called after instantiation it will return immediately.

Arguments

instId

Type: NumberU64

Optional. Clients must omit this argument or specify 0 because this is a global function.

Return value

Function has no return value.

5.21.11 InstantiationError

InstantiationError members:
code
Type: String
Error or warning code as a string symbol. This is one of the following:

License checking errors and warnings:
• "error_license_found_but_expired".
• "error_license_not_found".
• "error_license_count_exceeded".
• "error_cannot_contact_license_server".
• "warning_license_will_expire_soon".
• "error_general_license_error": For other license-related errors.

Parameter-related errors:
• "error_parameter_type_mismatch": Specified a String value for a non-String parameter or vice versa.
• "error_parameter_value_invalid": The value for a parameter is invalid.
• "error_unknown_parameter": Parameter name is unknown.
• "error_general_parameter_error": For other parameter-related errors.

General errors or warnings:
• "error_general_error": For all other errors.
• "error_general_warning": For all other warnings.

message
Type: String
Free-format error or warning message, potentially multiple lines long. This must repeat the error or warning reason given in code.

parameterName
Type: String
Optional. Optional. Name of the offending parameter for parameter-related errors. Mandatory for parameter-related errors. Must not be present for other errors or warnings.

severity
Type: String
Severity of the error or warning. This is one of the following:

error
This is an error. The simulation was not instantiated because of this error (but potentially also because of other errors).

warning
This is a warning. Warnings do not prevent the simulation from being instantiated but they might provide useful information to the user about potential problems.

5.21.12 InstantiationParameterValue

InstantiationParameterValue members:

name
Type: String
Hierarchical name of the parameter to set. This is the same as the ResourceInfo.name on page 5-80 value returned by simulation_getInstantiationParameterInfo() on page 5-191 or plugin_getInstantiationParameterInfo() on page 5-197.
value

Type: Value

Value of the parameter. Type is either a String for string parameters, or NumberU64[] for numeric parameters.

5.21.13 InstantiationResult

InstantiationResult members:

errors

Type: InstantiationError[]

List of errors and warnings that occurred during instantiation. See InstantiationError on page 5-193.

success

Type: Boolean

If True, the simulation was instantiated successfully. In this case, errors is either empty or only contains warnings. If False, the simulation was not instantiated. In this case, errors contains at least one error.

5.21.14 SimulationTimeObject

SimulationTimeObject members:

ticks

Type: NumberU64

Current simulation time in ticks. One tick is 1/tickHz seconds long. The elapsed simulation time is ticks/tickHz seconds.

tickHz

Type: NumberU64

Time resolution of the ticks value in Hz. For example, 1000 means that 1 tick = 1 ms.

running

Type: Boolean

Iff True, the simulation time is running, else it is stopped. Note that this information can already be outdated when the caller receives the response. When multiple simulation controllers start and stop the simulation, for example when multiple debuggers are connected, there is no way to reliably know whether the simulation is currently running or stopped. In this case, this is just a hint.

5.21.15 Event source IRIS_SIMULATION_SHUTDOWN_ENTER

This global event is generated when the simulation is about to enter its shutdown procedure. This is the earliest point at which instances can know that the simulation is about to exit. This event source has no fields.
If the event receiver activated this event source with `syncEc=True`, the shutdown procedure is not entered until the `ec_FOO()` function returns. This enables clients to perform last-minute operations, for example reading the final state of registers.

--- Note ---

The global instance might impose a global timeout for progressing with the shutdown sequence, to handle stalled or blocked clients. The shutdown sequence should only be paused for a few milliseconds, and as a guideline, not for more than 1000ms.

Instances must not unregister themselves from the instance registry using `instanceRegistry_unregisterInstance()` in response to this event, because other instances might want to continue to communicate with them during the shutdown phase.

If this event was requested with `syncEc=True`, the requesting instance should not make any Iris calls after returning from `ec_FOO()`. This is possible in a race-free way. If this event was requested with `syncEc=False`, the requesting instance should not make any Iris calls after this event was received. This is inherently racy. Any Iris calls made while or after this event was received with `syncEc=False` might return `E_unknown_instance_id`.

For in-process instances, the C++ `IrisInterface` pointers of the instance and of the global instance stay valid and can be used even when returning from this event.

**Related references**

5.17.5 `ec_FOO()` on page 5-148  
5.20.13 `instanceRegistry_unregisterInstance()` on page 5-180

### 5.21.16 Event source IRIS_SIMULATION_SHUTDOWN_LEAVE

This global event is generated when the simulation shutdown procedure is complete. After receiving this event, instances cannot communicate with each other. This event source has no fields.

This event is issued only after all instances that requested `IRIS_SIMULATION_SHUTDOWN_ENTER` with `syncEc=True` have returned from their `ec_FOO()` callback.

Instances can consider themselves to have been unregistered from the instance registry when they receive this event, so they should not call `instanceRegistry_unregisterInstance()` after receiving it. They are guaranteed not to receive any more Iris calls or responses after receiving it.

This event can be used by instances to destroy themselves. The C++ `IrisInterface` pointers must no longer be used after returning from `IrisInterface::irisHandleMessage()`. This is always possible race-free.

**Related references**

5.17.5 `ec_FOO()` on page 5-148
5.22 Plug-in loading and instantiation API

Plug-in instantiation is similar to simulation instantiation but with the following differences:

- Plug-ins can be instantiated more than once. When plugin_instantiate() is called on a plug-in factory, it creates an instance of the plug-in. The same factory instance can be used to create multiple instances of the plug-in with the same or different parameter values.
- The parameter names that are given by plugin_getInstantiationParameterInfo() are not hierarchical, but are relative to the plug-in instance. They are the same as the parameter names returned by calling resource_getList() on the plug-in instance.

This section contains the following subsections:

- 5.22.1 plugin_getInstantiationParameterInfo() on page 5-197.
- 5.22.2 plugin_instantiate() on page 5-197.
- 5.22.3 plugin_load() on page 5-198.

5.22.1 plugin_getInstantiationParameterInfo()

Gets the list of ParameterInfo used to instantiate instances of a plug-in.

**Arguments**

- instId
  
  Type: NumberU64
  
  Opaque number uniquely identifying the target instance.

**Return value**

ResourceInfo[]

ResourceInfo for the instantiation parameters of the plug-in.

**Errors**

- E_unknown_instance_id.

5.22.2 plugin_instantiate()

Instantiates a plug-in instance.

**Arguments**

- instName
  
  Type: String
  
  Optional. Used to construct the instance name for the new instance. The instance name is "client.plugin.<instName>". If omitted, the plug-in factory chooses a suitable instance name.

- paramValues
  
  Type: InstantiationParameterValue[]
  
  Optional. List of instantiation parameter values to use when instantiating the plug-in instance. Any parameters can be omitted and the paramValues argument can be omitted entirely. Any parameters that do not have a value set use their default value.

**Return value**

InstantiationResult

Indicates whether instantiation was successful and lists any errors and warnings that occurred during instantiation. See InstantiationResult on page 5-195.
5.22.3 plugin_load()

Loads a plug-in library.

Arguments

path

Type: String
Path to plug-in library to be loaded.

Return value

InstanceInfo[]
List of InstanceInfo for instances registered by this plug-in.

Errors

• E_error_loading_plugin.

Related references

5.6.11 resource_getList() on page 5-75
5.23 TCP server management API

The global instance manages the Iris TCP server.

The TCP server can be started or stopped at any time during the simulation but typically is started at the beginning of the simulation and stopped automatically when the simulation is shut down. If it is stopped while there are remote instances still connected to it, all those instances are automatically unregistered.

This section contains the following subsections:
• 5.23.1 tcpServer_getPort() on page 5-199.
• 5.23.2 tcpServer_start() on page 5-199.
• 5.23.3 tcpServer_stop() on page 5-199.
• 5.23.4 service_connect() on page 5-200.
• 5.23.5 service_disconnect() on page 5-200.

5.23.1 tcpServer_getPort()

Gets the TCP port number that the Iris server is listening on.

Return value
NumberU64
TCP port number that the server is listening on.

Errors
• E_server_not_running.

5.23.2 tcpServer_start()

Starts the Iris server listening on a port in the range [minPort to maxPort].

Arguments
maxPort
Type: NumberU64
Maximum port number to listen on.

minPort
Type: NumberU64
Minimum port number to listen on.

Return value
NumberU64
TCP port number that the server is listening on.

Errors
• E_server_already_running.
• E_data_size_error.

5.23.3 tcpServer_stop()

Stops the running Iris server. Any instances connected using TCP are automatically unregistered in the process.

Return value
Function has no return value.
Errors
• E_server_not_running.

5.23.4 service_connect()
Connects to a local or remote Iris service server.

Arguments
hostname
Type: String
Hostname of IrisService to connect to.
port
Type: NumberU64
Port of IrisService to connect to.

Return value
Function does not return a value.

Errors
• E_already_connected.
• E_unknown_hostname.
• E_socket_error.
• E_not_compatible.
• E_connection_refused.
• E_timeout.

5.23.5 service_disconnect()
Disconnects from an already connected Iris service server.

Arguments
hostname
Type: String
Hostname of the service to disconnect from.
port
Type: NumberU64
Port of the service to disconnect from.

Return value
Function does not return a value.

Errors
• E_socket_error.
• E_not_connected.
5.24 Checkpointing API

The Checkpointing API allows you to save and restore the state of an instance.

Note

Not all Iris components support checkpointing in this release.

This section contains the following subsections:

• 5.24.1 checkpoint_save() on page 5-201.
• 5.24.2 checkpoint_restore() on page 5-201.

5.24.1 checkpoint_save()

Saves a checkpoint on a specific instance.

Arguments

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

checkpointDir
Type: String
Directory to which the checkpoint is saved. This directory is always on the machine on which the target instance is running, which is not necessarily the machine on which the caller of this function is running.

Return value
Function has no return value.

Errors

• E_error_saving_checkpoint.

5.24.2 checkpoint_restore()

Restores a checkpoint on a specific instance.

Arguments

instId
Type: NumberU64
Opaque number uniquely identifying the target instance.

checkpointDir
Type: String
Directory from which the checkpoint is restored.

Return value
Function has no return value.

Errors

• E_error_restoring_checkpoint.
Chapter 6
Response error codes

This chapter describes the error codes that Iris defines. These are in addition to those defined in the JSON-RPC 2.0 specification.

Function calls in Iris are made according to the JSON-RPC 2.0 specification. This specification requires that a function call must return a response object, which must either contain a `result` member, if no error occurred, or an `error` object, if an error occurred.

Error objects have the following mandatory members:
- code
- message

It contains the following sections:
- 6.1 Function-specific error codes on page 6-203.
- 6.2 Function-independent error codes on page 6-208.
### 6.1 Function-specific error codes

The following table describes error codes that might be returned by specific functions only. These error codes are listed in the Errors section for each function that might return them.

<table>
<thead>
<tr>
<th>Error code</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_unknown_resource_id</td>
<td>0xE101</td>
<td>The rscId argument does not specify an existing resource for the specified target instance.</td>
</tr>
<tr>
<td>E_data_size_error</td>
<td>0xE102</td>
<td>The data argument, or one of the data arguments, if there are more than one, does not match the size that is expected by that function. Either the function expects data with a certain minimum or maximum size, or the size of an array or string is inconsistent with other parameters.</td>
</tr>
<tr>
<td>E_unknown_memory_space_id</td>
<td>0xE103</td>
<td>The spaceId argument does not specify a valid memory space for the target instance.</td>
</tr>
<tr>
<td>E_unknown_event_source</td>
<td>0xE104</td>
<td>The name argument does not specify a valid event source for the target instance.</td>
</tr>
<tr>
<td>E_unknown_event_field</td>
<td>0xE105</td>
<td>The field or fields argument contains an unknown event field for the specified event source.</td>
</tr>
<tr>
<td>E_unknown_event_stream_id</td>
<td>0xE106</td>
<td>The esId argument does not specify a valid event stream id, that is, one that was previously created with eventStream_create().</td>
</tr>
<tr>
<td>E_unknown_disassembly_mode</td>
<td>0xE107</td>
<td>The mode argument specifies an unknown disassembly mode.</td>
</tr>
<tr>
<td>E_unknown_callback_instance_id</td>
<td>0xE108</td>
<td>The callback instance id, ecInstId, is unknown.</td>
</tr>
<tr>
<td>E_syncEc_mode_not_supported</td>
<td>0xE109</td>
<td>The specified event source or function does not support synchronous callbacks.</td>
</tr>
<tr>
<td>E_counter_mode_not_supported</td>
<td>0xE10A</td>
<td>An unknown event counter id was passed to an event counter function.</td>
</tr>
<tr>
<td>E_unknown_file_descriptor</td>
<td>0xE10B</td>
<td>An unknown file descriptor was passed to the semihosting_provideInputData() function.</td>
</tr>
<tr>
<td>E_invalid_context</td>
<td>0xE10C</td>
<td>The function must not be called in this context. For example, the semihosting_return() function must only be called from within synchronous invocations of the ec_FOO() callback for the IRIS_SEMIHOSTING_CALL_EXTENSION event source.</td>
</tr>
<tr>
<td>E_unknown_sync_level</td>
<td>0xE10D</td>
<td>Unknown sync level value.</td>
</tr>
<tr>
<td>E_invalid_instName</td>
<td>0xE10E</td>
<td>The instance name is invalid. Either the class, that is, the first part of the hierarchical instance name, is unknown, or the dots separating the hierarchy levels have caused a formatting error.</td>
</tr>
<tr>
<td>E_instance_already_registered</td>
<td>0xE10F</td>
<td>The instance could not be registered because another instance with the same instName was already registered.</td>
</tr>
<tr>
<td>E_address_out_of_range</td>
<td>0xE113</td>
<td>The specified address is outside of the range minAddr to maxAddr for the specified memory space.</td>
</tr>
<tr>
<td>E_unknown_resource_group</td>
<td>0xE114</td>
<td>The specified resource group name is unknown.</td>
</tr>
<tr>
<td>E_unknown_event_source_id</td>
<td>0xE115</td>
<td>The specified event source id is unknown.</td>
</tr>
<tr>
<td>Error code</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>E_io_error</td>
<td>0xE116</td>
<td>An operation on a file or a socket on the callee side returned an I/O error, for example for <code>image_loadFile()</code>. End-of-file never results in E_io_error.</td>
</tr>
<tr>
<td>E_unknown_image_format</td>
<td>0xE117</td>
<td>The format of an image passed to <code>image_loadFile()</code> or <code>image_loadData()</code> is not recognized so it cannot be loaded.</td>
</tr>
<tr>
<td>E_image_format_error</td>
<td>0xE118</td>
<td>The format of an image passed to <code>image_loadFile()</code> or <code>image_loadData()</code> is recognized, but the image is either malformatted or contains unsupported constructs.</td>
</tr>
<tr>
<td>E_error_opening_file</td>
<td>0xE119</td>
<td>The file passed to <code>image_loadFile()</code> cannot be opened for reading.</td>
</tr>
</tbody>
</table>
| E_not_supported_for_event_source | 0xE11B| One of the following:  
  • A trace range or trace start point or stop point was set on an event source that does not support this feature.  
  • The trace range does not support the selected aspect.  
  • The event source does not support `eventStream_getState()`.
| E_not_a_counter            | 0xE11C| `eventStream_getCounter()` was called on an event stream that was not started as a counter, in other words, it was created with `eventStream_create(counter = False)`.
| E_unaligned_access         | 0xE11D| `memory_read()` or `memory_write()` was called with a start address that is not naturally aligned to the `byteWidth` argument.                                                                                                                                                                                                             |
| E_unsupported_attribute_name | 0xE11E| `memory_read()` or `memory_write()` was called with an unknown or unsupported attribute name in the `attrib` argument.                                                                                                                                                                                                                     |
| E_unsupported_attribute_value | 0xE11F| `memory_read()` or `memory_write()` was called with a valid attribute name, but the value specified is unsupported or invalid, regardless of the values of other attributes, or has the wrong type.                                                                                                                                         |
| E_unsupported_attribute_combination | 0xE120| `memory_read()` or `memory_write()` was called with a combination of attributes that does not make sense or is otherwise unsupported. Note: Target instances are not required to detect all invalid attribute combinations.                                                                                                 |
| E_unsupported_breakpoint_type       | 0xE121| `breakpoint_set()` was called with an unknown or unsupported breakpoint type.                                                                                                                                                                                                                                                        |
| E_unsupported_argument_combination | 0xE122| A function was called with a combination of optional arguments that is either unsupported or does not make sense. This includes missing optional arguments that are mandatory under some circumstances.                                                                                                                                         |
| E_invalid_rwMode           | 0xE123| `breakpoint_set()` was called with an `rwMode` argument that is not "r", "w", or "rw".
| E_unknown_breakpoint_id    | 0xE124| The specified breakpoint id, bptId, is not known by the instance.                                                                                                                                                                                                                                                                          |
| E_unsupported_translation  | 0xE125| `memory_translateAddress()` was called with an unsupported pair of memory spaces. Models usually only support translations for very specific pairs of memory spaces and only in certain directions, for example only virtual to physical.                                                                                   |
| E_unit_not_supported       | 0xE126| A step function was called with a unit that is not supported by the instance.                                                                                                                                                                                                                                                        |
### Table 6-1 Function-specific error codes (continued)

<table>
<thead>
<tr>
<th>Error code</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_unknown_function_name</td>
<td>0xE127</td>
<td>A function name was passed as an argument to another function, but the function is not known by the associated instance. This error is different to E_function_not_supported_by_instance in that the function in question is not the function currently being called, but is referred to by other means.</td>
</tr>
<tr>
<td>E_writing_init_time_parameter</td>
<td>0xE128</td>
<td>resource_write() was called on an initialization-time parameter, which can only be set before simulation instantiation and are read-only after that.</td>
</tr>
<tr>
<td>E_not_connected</td>
<td>0xE129</td>
<td>The client, for example IrisTcpClient, is not connected to a simulation and so the call cannot be sent anywhere. This is only returned by IrisTcpClient or similar clients.</td>
</tr>
<tr>
<td>E_index_out_of_range</td>
<td>0xE12A</td>
<td>The specified index is out of range, for example for table_read().</td>
</tr>
<tr>
<td>E_unknown_table_id</td>
<td>0xE12B</td>
<td>A table function was called with an unknown table id.</td>
</tr>
<tr>
<td>E_unsupported_mode</td>
<td>0xE12E</td>
<td>A function was called with an unknown or unsupported mode argument.</td>
</tr>
<tr>
<td>E_unknown_instance_name</td>
<td>0xE12F</td>
<td>Function instanceRegistry_getInstanceInfoByName() was called with an unknown instance name. The class of the instance must be part of the instance name, for example component._cluster_0 instead of just cluster_0. Also, the presence of component._a._b._c does not guarantee the presence of component._a._b.</td>
</tr>
<tr>
<td>E_connection_refused</td>
<td>0xE130</td>
<td>Tried to connect to a server but there is no Iris server running on that port.</td>
</tr>
<tr>
<td>E_timeout</td>
<td>0xE131</td>
<td>A function that supports a timeout was called and it timed out.</td>
</tr>
<tr>
<td>E_unknown_hostname</td>
<td>0xE132</td>
<td>Tried to connect to a host using a hostname, but the hostname is not known.</td>
</tr>
<tr>
<td>E_socket_error</td>
<td>0xE133</td>
<td>A low level read, write, or configuration operation failed on the TCP socket.</td>
</tr>
<tr>
<td>E_thread_error</td>
<td>0xE134</td>
<td>A low-level threading function failed.</td>
</tr>
<tr>
<td>E_not_compatible</td>
<td>0xE135</td>
<td>Tried to connect to a server, but the server is not compatible with this client.</td>
</tr>
<tr>
<td>E_already_connected</td>
<td>0xE136</td>
<td>Tried to connect to a server but the client is already connected to one. Disconnect the client first.</td>
</tr>
<tr>
<td>E_error_reading_write_only_resource</td>
<td>0xE137</td>
<td>resource_read() returns this error code in the error member of the returned ResourceReadResult for write-only resources that were read. resource_read() does not return it as the error code. Note It is valid, and encouraged, for registers that are architecturally write-only to return a useful value when read through resource_read(). resource_read() is not an architectural register read and does not have to follow architectural rules, including architectural semantics and permissions.</td>
</tr>
<tr>
<td>Error code</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>E_error_writing_read_only_resource</td>
<td>0xE138</td>
<td><code>resource_write()</code> returns this error code in the <code>error</code> member of the returned <code>ResourceWriteResult</code> for read-only resources that were written. <code>resource_write()</code> does not return it as the error code. Note: Some registers that are architecturally read-only might be writable through <code>resource_write()</code> because <code>resource_write()</code> is not an architectural write.</td>
</tr>
<tr>
<td>E_error_reading_resource</td>
<td>0xE139</td>
<td><code>resource_read()</code> returns this error code in the <code>error</code> member of the returned <code>ResourceReadResult</code> when the value of a resource cannot be determined, or it produced an error that cannot be represented in a better way. <code>resource_read()</code> does not return it as the error code. This is a catch-all error for resource reads that failed.</td>
</tr>
<tr>
<td>E_error_writing_resource</td>
<td>0xE13A</td>
<td><code>resource_write()</code> returns this in the <code>error</code> member of the returned <code>ResourceWriteResult</code> for resources that could not be written, or writing it produced an error. <code>resource_write()</code> does not return it as the error code. This is a catch-all error for resource writes that failed.</td>
</tr>
<tr>
<td>E_operation_interrupted</td>
<td>0xE13C</td>
<td>An operation was interrupted by an end user. This is usually only returned by specific callbacks that are implemented by clients to allow chunked transfers of data. This error code tells the original operation, for example <code>image_loadDataPull()</code>, to stop. The interrupted operation in turn returns <code>E_operation_interrupted</code>.</td>
</tr>
<tr>
<td>E_error_setting_breakpoint</td>
<td>0xE13D</td>
<td>A breakpoint could not be set because the breakpoint logic that is internal to a specific instance failed.</td>
</tr>
<tr>
<td>E_error_deleting_breakpoint</td>
<td>0xE13E</td>
<td>A breakpoint could not be deleted because the breakpoint logic that is internal to a specific instance failed.</td>
</tr>
<tr>
<td>E_error_creating_event_stream</td>
<td>0xE13F</td>
<td>An event stream could not be created because the event logic that is internal to the instance failed.</td>
</tr>
<tr>
<td>E_error_enabling_event_stream</td>
<td>0xE140</td>
<td>An event stream could not be enabled because the event logic that is internal to the instance failed.</td>
</tr>
<tr>
<td>E_error_disabling_event_stream</td>
<td>0xE141</td>
<td>An event stream could not be destroyed because the event logic that is internal to the instance failed.</td>
</tr>
<tr>
<td>E_error_memory_abort</td>
<td>0xE142</td>
<td>Only returned in the <code>error</code> field in <code>MemoryReadResult</code> or <code>MemoryWriteResult</code>. The memory value could not be read because the memory subsystem aborted the operation. A load instruction from this address would also have failed. The corresponding bits in <code>data</code> should be ignored.</td>
</tr>
<tr>
<td>E_approximation</td>
<td>0xE143</td>
<td>Only returned in the <code>error</code> field in <code>MemoryReadResult</code> or <code>ResourceReadResult</code>. The memory value could be read but is only an approximation, for example because one of the instances is not in a debuggable state. A load instruction from this address would have succeeded.</td>
</tr>
<tr>
<td>Error code</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>E_value_not_available</td>
<td>0xE144</td>
<td>Only returned in the error field in MemoryReadResult or ResourceReadResult. The memory value is currently not available and cannot even be approximated. A load instruction from this address would have succeeded. For example, this can happen when an instance is not in a debuggable state.</td>
</tr>
<tr>
<td>E_current_thread_does_not_have_a_message_queue</td>
<td>0xE145</td>
<td>Returned by IrisC_processAsyncMessages() functions when called from a thread that is not synchronous to the simulation thread.</td>
</tr>
<tr>
<td>E_server_already_running</td>
<td>0xE146</td>
<td>Returned by tcpServer_start() when a server is already running.</td>
</tr>
<tr>
<td>E_server_not_running</td>
<td>0xE147</td>
<td>Returned by tcpServer_getPort() or tcpServer_stop() if no server is running.</td>
</tr>
<tr>
<td>E_error_loading_dso</td>
<td>0xE148</td>
<td>Could not load a DSO. This might be because the specified file could not be found or it was not a valid DSO.</td>
</tr>
<tr>
<td>E_invalid_plugin</td>
<td>0xE149</td>
<td>Returned by plugin_load() if the plug-in path does not refer to a valid Iris plug-in.</td>
</tr>
<tr>
<td>E_plugin_already_loaded</td>
<td>0xE14A</td>
<td>Returned by plugin_load() if the plug-in path refers to a plug-in that has already been loaded.</td>
</tr>
<tr>
<td>E_error_writing_read_only_cell</td>
<td>0xE14B</td>
<td>Returned when trying to write a read only table cell.</td>
</tr>
<tr>
<td>E_invalid_breakpoint_condition</td>
<td>0xE14C</td>
<td>One or more breakpoint condition was invalid. This might be because the condition was not recognized, or it was the wrong type, or it was not applicable to the breakpoint type being set.</td>
</tr>
<tr>
<td>E_full_reset_not_supported</td>
<td>0xE14D</td>
<td>Not all parts of the simulation could be reset by the SimulationEngine. It might still be possible to partially reset the simulation by calling simulation_reset() with allowPartialReset=true.</td>
</tr>
<tr>
<td>E_simulation_running</td>
<td>0xE14E</td>
<td>The request cannot be carried out while the simulation is running. The caller must stop the simulation and try again.</td>
</tr>
<tr>
<td>E_function_already_registered</td>
<td>0xE14F</td>
<td>A function with the same name is already registered in an instance.</td>
</tr>
<tr>
<td>E_event_source_already_registered</td>
<td>0xE150</td>
<td>An event source with the same name is already registered in an instance.</td>
</tr>
<tr>
<td>E_error_saving_checkpoint</td>
<td>0xE151</td>
<td>checkpoint_save() was unable to save the checkpoint, for example because a directory was not writable.</td>
</tr>
<tr>
<td>E_error_restoring_checkpoint</td>
<td>0xE152</td>
<td>checkpoint_restore() was unable to restore the checkpoint, for example because a file was not found.</td>
</tr>
<tr>
<td>E_unsupported_option_name</td>
<td>0xE153</td>
<td>An unsupported option name was specified in an options argument.</td>
</tr>
<tr>
<td>E_unsupported_option_value</td>
<td>0xE154</td>
<td>An unsupported option value was specified in an options argument.</td>
</tr>
<tr>
<td>E_unsupported_option_combination</td>
<td>0xE155</td>
<td>An unsupported combination of optional values was specified in an options argument.</td>
</tr>
</tbody>
</table>
6.2 Function-independent error codes

The following table describes the error codes that any function might return. These error codes are not listed in the Errors sections, for brevity.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E_u64json_encoding_error</td>
<td>-0xB0</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• An invalid 64-bit code was found while parsing a U64JSON value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The explicit size of an array, which was used to skip the array, is inconsistent with the array contents.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The explicit size of an object, which was used to skip the object, is incorrect.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Duplicate keys appear in an object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The container length exceeds the length of the available data or the container length is shorter than the available data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A number as a string has a parse error.</td>
</tr>
<tr>
<td>E_malformatted_request</td>
<td>-0xB1</td>
<td>A malformed request was received. For example, the params member is an array instead of an object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This error is only returned when the request is sufficiently well-formatted to identify that it is a request, for example it contains a method member, and to enable the return path to be traced back.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requests that contain an invalid or missing method, instId, or request id might not be able to respond to the caller.</td>
</tr>
<tr>
<td>E_malformatted_response</td>
<td>-0xB2</td>
<td>A malformed response was received. For example, this is returned when a response was received that contains both an error member and a result member.</td>
</tr>
<tr>
<td>E_ok</td>
<td>0</td>
<td>Indicates no error. This is never returned by JSON RPC 2.0 calls because they only return an error object if an error occurred. This value is reserved for the OK case for C++ interfaces or for variables that need to hold a non-error state.</td>
</tr>
<tr>
<td>E_no_response_yet</td>
<td>1</td>
<td>No error. No response was received yet for a request. This is never returned by Iris functions and is only used in the Iris framework state machines.</td>
</tr>
<tr>
<td>E_unknown_instance_id</td>
<td>0xE100</td>
<td>The instId argument does not specify an existing target instance.</td>
</tr>
<tr>
<td>E_function_not_supported_by_instance</td>
<td>0xE110</td>
<td>A function was called with a valid instId argument, but the instance does not support the function that was called. This error applies only to the function name, not to arguments or their values. This error is also returned when an unsupported global function, without an instId, was called.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>E_unsupported_argument_name</td>
<td>0xE111</td>
<td>A function received one or more argument names that it does not support. This error tells the caller that the expected functionality is not available in the function. The caller can fall back to a simpler function call with fewer parameters, if it is designed to do so. The callee must put an object {&quot;name&quot;: X} into <code>error.data</code> of the response object, where X is one of the unsupported argument names. This does not imply that other argument names are valid.</td>
</tr>
<tr>
<td>E_unsupported_argument_value</td>
<td>0xE112</td>
<td>A function received an unsupported value for one or more arguments. This is a catch-all error code, which should only be used if a more specific error code is not available. The callee must put an object {&quot;name&quot;: X} into <code>error.data</code> of the response object, where X is the name of one of the arguments containing an unsupported value. This does not imply that other argument values are valid.</td>
</tr>
<tr>
<td>E_missing_mandatory_argument</td>
<td>0xE11A</td>
<td>A function was called without a mandatory argument. The callee must put an object {&quot;name&quot;: X} into <code>error.data</code> of the response object, where X is one of the missing argument names. This does not imply that all other arguments are present.</td>
</tr>
<tr>
<td>E_argument_type_mismatch</td>
<td>0xE12C</td>
<td>A value of an incompatible type was specified for a function parameter. This includes types of object members that are passed into functions. See 3.1 JSON data types on page 3-24 for type compatibility and conversion rules. The callee must put an object {&quot;name&quot;: X} into <code>error.data</code> of the response object containing the parameter name of one of the errors encountered. This does not imply that other parameters are valid. It is undefined whether the request was ignored or partially completed. Clients should assume that the request was ignored and should repeat the request.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>E_not_supported_while_instance_is_blocked</td>
<td>0xE12D</td>
<td>A function $F$ was called by instance $A$ on instance $B$ while instance $B$ is blocked in a synchronous function call to instance $A$, for example in a synchronous <code>ec_FOO()</code> call. Instance $B$ cannot complete function $F$ while it is blocked in the call to instance $A$.</td>
</tr>
<tr>
<td>E_not_implemented</td>
<td>0xE13B</td>
<td>The function being called or the feature being requested or used is not implemented. A common use case is to return this in the error member of a <code>ResourceReadResult</code> or <code>ResourceWriteResult</code> for resources that are known to an instance but are not implemented. It might be more useful to return <code>E_not_implemented</code> rather than returning dummy values for partially implemented interfaces during development. Iris functions should generally not return this error for functionality that will never be implemented. <code>E_function_not_supported_by_instance</code> or the <code>E_*unsupported*</code> errors should be preferred if they are more appropriate.</td>
</tr>
</tbody>
</table>