Iris

Version 1.0

Developer Guide

arm

101196_0100_00_en

Iris

Developer Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	23 November 2018	Non-Confidential	New document.

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Web Address

http://www.arm.com

Contents
Iris Developer Guide

Preface

About this book 7
References 9
Terms and abbreviations 10

Chapter 1 Iris overview

1.1 Overview 1-12
1.2 Interfaces and communication 1-13

Chapter 2 Generic function call interface

2.1 JSON data types 2-17
2.2 JSON-RPC 2.0 function call format 2-19
2.3 Synchronous and asynchronous behavior 2-21
2.4 Sending a request, a notification, and a response 2-22
2.5 U64JSON 2-23
2.6 Function call optimizations 2-31
2.7 IrisC interface 2-32
2.8 IrisRpc (RPC transport layer) 2-41
2.9 JSON-RPC 2.0 over HTTP 2-44
2.10 Threading model and ordering 2-45

Chapter 3 Object model

3.1 Object model overview 3-48
3.2 Instances 3-49

Chapter 4 Iris APIs

4.1 Iris API documentation 4-52
4.2 Naming conventions 4-53
4.3 Resources 4-54
4.4 Memory 4-59
4.5 Disassembly 4-66
4.6 Tables 4-67
4.7 Image loading and saving 4-68
4.8 Simulation time execution control 4-70
4.9 Debuggable state 4-72
4.10 Stepping 4-75
4.11 Per-instance execution control 4-77
4.12 Breakpoints 4-78
4.13 Notification and discovery of state changes 4-81
4.14 Events and trace interface 4-82
4.15 Semihosting 4-87
4.16 Simulation accuracy (sync levels) 4-92
4.17 Checkpointing 4-94
4.18 Instance registry, instance discovery, and interface discovery 4-95
4.19 Simulation instantiation and discovery 4-102
4.20 Plug-in loading and instantiation 4-105
4.21 Iris-text-format 4-106
4.22 instId argument 4-109
4.23 Compatibility rules for function callers and callees 4-110
4.24 TCP server management 4-111

Preface

This preface introduces the *Iris Developer Guide*.

It contains the following:

- *About this book* on page 7.
- *References* on page 9.
- *Terms and abbreviations* on page 10.

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 the Iris interface for debug and trace.

Chapter 2 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 3 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 4 Iris APIs

This chapter provides background and conceptual information for the Iris APIs.

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_00_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*.

References

This document refers to the following websites:

URL	Title
http://json.org/index.html	Introducing JSON
http://www.jsonrpc.org/specification	JSON-RPC 2.0 Specification
http://www.simple-is-better.org/json-rpc/transport_http.html	JSON-RPC 2.0 Transport: HTTP
https://tools.ietf.org/html/rfc7230	RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing

Terms and abbreviations

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

Term	Description
iff	Short for *if and only if*.
IPC	Inter-Process Communication, either on the same host, for example TCP or pipes, or on different hosts, for example TCP.
DSO	Dynamic Shared Object (`*.so`) or DLL.
JSON	JavaScript Object Notation. A compact textual representation of data.
JSON RPC	Remote Procedure Call protocol using JSON.
U64JSON	Binary JSON variant that is based on arrays of 64-bit values. It is defined in the *Iris Developer Guide*.
Component	A piece of software with well-defined abstract interfaces that represents a piece of hardware or other functionality.
Target	Sometimes used as a synonym for *component* or *instance* but generally avoided in this document.
Instance	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.
Event	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.
Iris	An interface for debug and trace. Not an abbreviation or acronym.

Chapter 1
Iris overview

This chapter describes the purpose and implementation of the Iris interface for debug and trace.

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 1-12.
- *1.2 Interfaces and communication* on page 1-13.

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, `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 `uint64_t` values, in-process to remove the JSON parsing overhead.

1.2 Interfaces and communication

The following diagram provides an overview of the Iris interfaces:

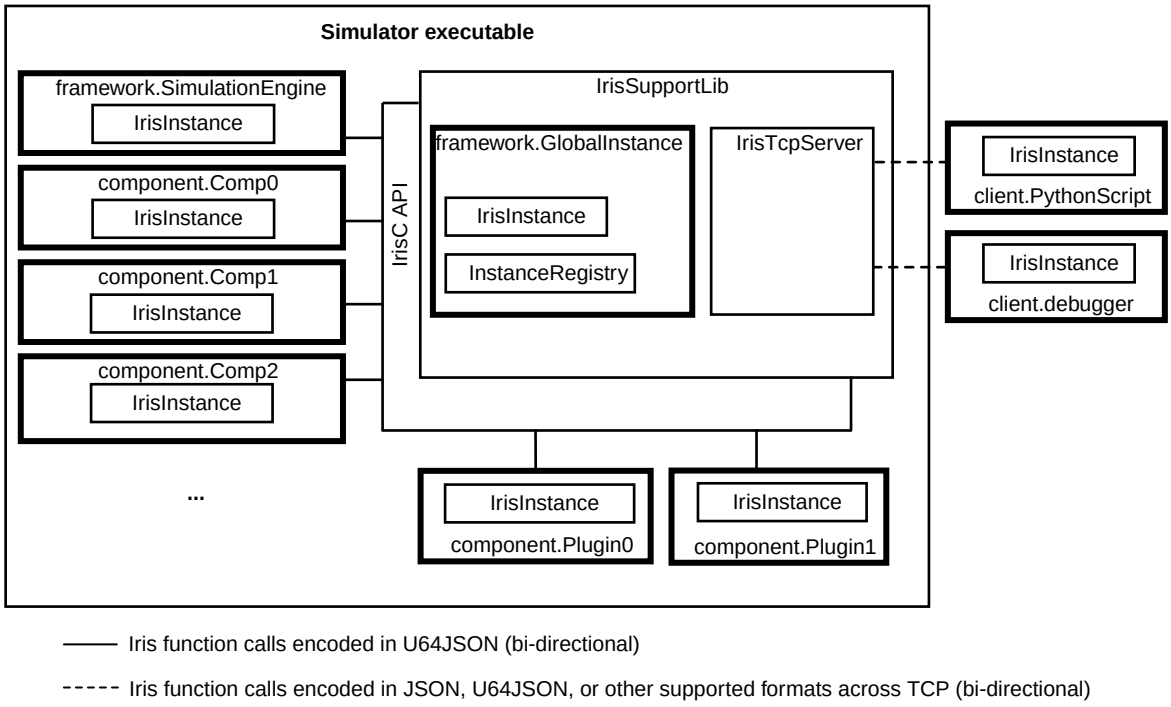

Figure 1-1 Iris architecture

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.

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.

Chapter 2
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:

- *2.1 JSON data types* on page 2-17.
- *2.2 JSON-RPC 2.0 function call format* on page 2-19.
- *2.3 Synchronous and asynchronous behavior* on page 2-21.
- *2.4 Sending a request, a notification, and a response* on page 2-22.
- *2.5 U64JSON* on page 2-23.
- *2.6 Function call optimizations* on page 2-31.
- *2.7 IrisC interface* on page 2-32.
- *2.8 IrisRpc (RPC transport layer)* on page 2-41.
- *2.9 JSON-RPC 2.0 over HTTP* on page 2-44.
- *2.10 Threading model and ordering* on page 2-45.

2.1 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 2-1 Implicit type conversions

Original value	To Number	To NumberU64	To NumberS64	To Boolean	To Object	To Array	To String
Number	✓	-/✓ (Round)(Range)	-/✓ (Round)(Range)	-/✓ (Round)(only 1/0)	-	-	-
NumberU64	✓	✓	-/✓ (Range)	-/✓ (only 1/0)	-	-	-
NumberS64	✓	-/✓ (Range)	✓	-/✓ (only 1/0)	-	-	-
Boolean	✓ (to 1/0)	✓ (to 1/0)	✓ (to 1/0)	✓	-	-	-
Object	-	-	-	-	✓	-	-
Array	-	-	-	-	-	✓	-
String	-	-	-	-	-	-	✓

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.

2.2 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 *3.2 Instances* on page 3-49. 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 3 Object model* on page 3-47

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.

2.3 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.

2.4 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 *2.2 JSON-RPC 2.0 function call format* on page 2-19.

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 *2.2 JSON-RPC 2.0 function call format* on page 2-19. 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 *2.2 JSON-RPC 2.0 function call format* on page 2-19. 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.

2.5 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:

- *2.5.1 U64JSON format* on page 2-23.
- *2.5.2 Container length* on page 2-28.
- *2.5.3 Endianness* on page 2-29.
- *2.5.4 Signedness and integer representation* on page 2-29.
- *2.5.5 Numbers with arbitrary size and precision* on page 2-29.
- *2.5.6 Optimizations and normalized form* on page 2-29.
- *2.5.7 U64JSON examples* on page 2-30.

2.5.1 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 *2.5.7 U64JSON examples* on page 2-30.

Table 2-2 U64JSON format

MSB	JSON type	Meaning
`0x0`	`Number`	Positive integer Numbers from 0 to `0x0fffffffffffffff` (60 bits).
`0x1`	`Number`	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`.

Table 2-2 U64JSON format (continued)

MSB	JSON type	Meaning

0x2-0x7	String	String. Short string, 255 or fewer bytes long, where string[6] is in 0x20-0x7f if longer than 6 bytes. Format:
		`// nn = string length, s…+MM = string`
		`uint64_t msb_and_string_data = 0xMMss ssss ssss ssnn;`
		`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:
		`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:
		`// 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 2-2 U64JSON format (continued)

MSB	JSON type	Meaning

0xb	Object	Object container. Map from String to Value. Format:
		`// 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:
		`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:
		`uint64_t msb = 0xc100 0000 0000 0000;`
		`uint64_t number;`
		Container length: 2.

0xc2-0xc9	-	Reserved, except 0xc9fffffffffffff 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:
		`uint64_t msb = 0xca00 0000 0000 0000;`
		`double number;`
		Container length: 2.

0xcb	-	Reserved.

Table 2-2 U64JSON format (continued)

MSB	JSON type	Meaning

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;`
0xd	-	Reserved

Table 2-2 U64JSON format (continued)

MSB	JSON type	Meaning
`0xe0`	Request	Encodes an Iris request message.
		`uint64_t msb = 0xe0vv xxxx xxxx xxxx xxxx;` `vv` indicates the JSON-RPC version and is `0x20`. `x`… is the container length.

		`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.

		`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.

		`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.

		`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.

Table 2-2 U64JSON format (continued)

MSB	JSON type	Meaning
`0xe2`	Response	Encodes an Iris response message.
		`uint64_t msb = 0xe2vv xxxx xxxx xxxx xxxx;` `vv` indicates the JSON-RPC version and is `0x20`. `x...` is the container length.

		`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` and is not encoded according to the Number format in this table.

		`uint64_t instId; // response[2]`
		Indicates the destination instance, in other words, the caller of the Iris function. The `instId` is a plain `uint64` and is not encoded according to the Number format in this table.

		`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` and is not encoded according to the Number format in this table. See *Response error codes* in the *Iris Reference Manual* for information about error codes.

		`Value result; // response[4:n]`
		or
		`String message; // response[4:m]`
		The type and meaning of this field depends on the value of the preceeding `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.

		`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`.

`0xf`	`Number`	Positive integer Numbers from `0xf000000000000000`-`0xffffffffffffffff` (60 bits).

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 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.

2.5.6 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 $2^{64}-2^{60}$ to $2^{64}-1$ (`0xF000000000000000` to `0xFFFFFFFFFFFFFFFF`)
  MSB is `0xf`.

  Other positive 64-bit integers, in the range 2^{60} to $2^{64}-2^{60}-1$ (`0x1000000000000000` to `0xEFFFFFFFFFFFFFFF`)
  MSB is `0xc0`.

  Other negative 64-bit integers, in the range -2^{63} to $-2^{60}-1$
  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.

2.5.7 U64JSON examples

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

Table 2-3 U64JSON examples

JSON value	U64JSON representation
`[1,2,3]`	`0x8000000000000003, 1, 2, 3`
`["1",2,3]`	`0xa000000000000005, 3, 0x2000 0000 0000 3101, 2, 3`
`"abc"`	`0x2000 0000 6362 6103`
`""`	`0x2000000000000000`
`"numbyte"`	`0x6574 7962 6d75 6e07`
`"numbytes"`	`0x6574 7962 6d75 6e08, 0x73`
`0`	`0`
`1`	`1`
`0xffff ffff ffff ffff`	`0xffff ffff ffff ffff` This is $+2^{64}$-1, not -1.
`-1`	`0x1fff ffff ffff ffff`
`0xaabb ccdd eeff 0011`	`0xc000 0000 0000 0000, 0xaabb ccdd eeff 0011`
`-0x1234 5678 9012 3456`	`0xc100 0000 0000 0000, 0xedcb a987 6fed cbaa`
`{"num":1,"b":2,"c":3}`	`0xb000 0000 0000 0008, 3, 0x2000 0000 6d75 6e03, 1, 0x2000 0000 0000 6201, 2, 0x2000 0000 0000 6301, 3` **Note** 13 JSON tokens (26 chars) translate into 8 `uint64_t` values (64 bytes) in U64JSON.
`Null`	`0xcd00000000000000`
`False`	`0xce00000000000000`
`True`	`0xcf00000000000000`

2.6 Function call optimizations

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

This section contains the following subsections:

- *2.6.1 Fast argument parsing using sorted arguments* on page 2-31.
- *2.6.2 String comparison and hashing* on page 2-31.

2.6.1 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.

2.6.2 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).

2.7 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:

- *2.7.1 Memory and interface ownership* on page 2-32.
- *2.7.2 IrisCore lifecycle functions* on page 2-32.
- *2.7.3 IrisClient lifecycle functions* on page 2-32.
- *2.7.4 General IrisC functions* on page 2-33.
- *2.7.5 Plug-in API* on page 2-37.

2.7.1 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.

2.7.2 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.

2.7.3 IrisClient lifecycle functions

IrisSupportLib defines the following functions to manage its lifecycle:

IrisClient_connect()

```
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()

```
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.

2.7.4 General IrisC functions

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

handleMessage()

`handleMessage()` function.

```
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.

obtainInterface()

`obtainInterface()` function.

```
int64_t obtainInterface(void *obtain_interface_context, uint64_t instId, const char
*requested_interface, void **interface_ptr_out);
int64_t IrisCore_obtainInterface(void *iris_core_context, uint64_t instId, const char
*requested_interface, void **interface_ptr_out);
int64_t IrisClient_obtainInterface(void *iris_client_context, uint64_t instId, const char
*requested_interface, void **interface_ptr_out);
```

Retrieves a named native interface pointer for an instance

Arguments:

obtain_interface_context
The context pointer for the callee. For `IrisCore_obtainInterface()`, this is the `iris_core_context` pointer provided by `IrisCore_init()`. For `IrisClient_obtainInterface()`, this is the `iris_client_context` pointer provided by `IrisClient_connect()`. For other `obtainInterface()` functions, this should be the context pointer that is associated with that function.

instId
Instance Id of the destination instance.

requested_interface
Name of the native interface that the caller is requesting. This should be a null-terminated C string.

interface_ptr_out
`*interface_ptr_out` is set to the requested interface pointer.

Return value:

An `IrisErrorCode` value indicating whether the call was successful.

registerChannel()

`registerChannel()` function.

```
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()

unregisterChannel() function.

```
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.

```
struct IrisC_CommunicationChannel
{
    uint64_t CommunicationChannel_version;
    IrisC_HandleMessageFunction handleMessage_function;
    void *handleMessage_context;
    IrisC_ObtainInterfaceFunction obtainInterface_function;
    void *obtainInterface_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.

obtainInterface_function
obtainInterface() function pointer for this channel. This member can be NULL if the instance does not support native interfaces.

obtainInterface_context
Context pointer to pass when calling obtainInterface_function. If obtainInterface_function is NULL, obtainInterface_context should also be NULL.

processAsyncMessages()

`processAsyncMessages()` function.

```
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:

```
while (no_response_received)
    processAsyncMessages(context, IrisC_AsyncMessage_Blocking);
```

irisInitPlugin()

`irisInitPlugin()` function.

```
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.

```
struct IrisC_Functions
{
```

```
    uint64_t Functions_version;
    void *iris_c_context;
    IrisC_RegisterChannelFunction registerChannel_function;
    IrisC_UnregisterChannelFunction unregisterChannel_function;
    IrisC_HandleMessageFunction handleMessage_function;
    IrisC_ObtainInterfaceFunction obtainInterface_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.

`obtainInterface_function`
Pointer to an IrisC library `obtainInterface()` function.

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

2.7.5 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.

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. 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 2-4 Iris initialization phase callbacks and SystemC simulation phases

Iris initialization phase callback	SystemC simulation phase
IRIS_SIM_PHASE_INITIAL_PLUGIN_LOADING_COMPLETE	sc_main(), before sc_start()
IRIS_SIM_PHASE_INSTANTIATE_ENTER	before_end_of_elaboration()
IRIS_SIM_PHASE_INSTANTIATE	
IRIS_SIM_PHASE_INSTANTIATE_LEAVE	
IRIS_SIM_PHASE_INIT_ENTER	
IRIS_SIM_PHASE_INIT	
IRIS_SIM_PHASE_INIT_LEAVE	
IRIS_SIM_PHASE_BEFORE_END_OF_ELABORATION	
IRIS_SIM_PHASE_END_OF_ELABORATION	end_of_elaboration()
IRIS_SIM_PHASE_INITIAL_RESET_ENTER	start_of_simulation()
IRIS_SIM_PHASE_INITIAL_RESET	
IRIS_SIM_PHASE_INITIAL_RESET_LEAVE	
IRIS_SIM_PHASE_START_OF_SIMULATION	
IRIS_SIM_PHASE_RESET_ENTER	sc_start(), after start_of_simulation(), and before end_of_simulation()
IRIS_SIM_PHASE_RESET	
IRIS_SIM_PHASE_RESET_LEAVE	
IRIS_SIM_PHASE_END_OF_SIMULATION	end_of_simulation()
IRIS_SIM_PHASE_TERMINATE_ENTER	
IRIS_SIM_PHASE_TERMINATE	
IRIS_SIM_PHASE_TERMINATE_LEAVE	

2.8 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()`.

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 JSON RPC 2.0 server. An Iris server is also both a JSON RPC 2.0 client and JSON RPC 2.0 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:

- *2.8.1 IrisRpc connection handshake* on page 2-41.
- *2.8.2 Rejecting a connection request* on page 2-42.
- *2.8.3 Supported formats* on page 2-42.
- *2.8.4 IrisRpc message format* on page 2-43.
- *2.8.5 TCP considerations* on page 2-43.

2.8.1 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.

2.8.2 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.

2.8.3 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.

2.8.4 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.

2.8.5 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.

2.9 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:

- *2.9.1 Recognizing a session as JSON-RPC HTTP* on page 2-44.
- *2.9.2 Persistent connections* on page 2-44.

2.9.1 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.

2.9.2 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.

2.10 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:

- *2.10.1 Asynchronous functions* on page 2-45.
- *2.10.2 Reentrancy* on page 2-45.
- *2.10.3 Ordering rules for requests, notifications, and responses* on page 2-46.

2.10.1 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.

2.10.2 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`.

2.10.3 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.

Chapter 3
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:

- *3.1 Object model overview* on page 3-48.
- *3.2 Instances* on page 3-49.

3.1 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 *4.18.1 Hierarchical instance names and instance classes* on page 4-95.

3.2 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 *4.18 Instance registry, instance discovery, and interface discovery* on page 4-95 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 *4.18.6 Interface discovery* on page 4-99 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 *4.22 instId argument* on page 4-109.

Chapter 4
Iris APIs

This chapter provides background and conceptual information for the Iris APIs.

For complete and detailed reference information, see the *Iris Reference Manual*.

It contains the following sections:

- *4.1 Iris API documentation* on page 4-52.
- *4.2 Naming conventions* on page 4-53.
- *4.3 Resources* on page 4-54.
- *4.4 Memory* on page 4-59.
- *4.5 Disassembly* on page 4-66.
- *4.6 Tables* on page 4-67.
- *4.7 Image loading and saving* on page 4-68.
- *4.8 Simulation time execution control* on page 4-70.
- *4.9 Debuggable state* on page 4-72.
- *4.10 Stepping* on page 4-75.
- *4.11 Per-instance execution control* on page 4-77.
- *4.12 Breakpoints* on page 4-78.
- *4.13 Notification and discovery of state changes* on page 4-81.
- *4.14 Events and trace interface* on page 4-82.
- *4.15 Semihosting* on page 4-87.
- *4.16 Simulation accuracy (sync levels)* on page 4-92.
- *4.17 Checkpointing* on page 4-94.
- *4.18 Instance registry, instance discovery, and interface discovery* on page 4-95.
- *4.19 Simulation instantiation and discovery* on page 4-102.
- *4.20 Plug-in loading and instantiation* on page 4-105.
- *4.21 Iris-text-format* on page 4-106.
- *4.22 instId argument* on page 4-109.

- *4.23 Compatibility rules for function callers and callees* on page 4-110.
- *4.24 TCP server management* on page 4-111.

4.1 Iris API documentation

This book and the Iris Reference Manual use 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 *Iris Reference Manual* describes any Objects that are returned 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.

See *Iris Reference Manual* for descriptions of Iris interfaces and error codes.

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.

Related information

Iris Reference Manual

4.2 Naming conventions

This topic describes the naming conventions that the Iris interface uses.

Table 4-1 Naming conventions for Iris functions, objects, and events

Identifier	Case	Example
Function name	lowerCamelCase	`instanceRegistry_registerInstance`
Function argument name	lowerCamelCase	`instId`
Object member name	lowerCamelCase	`bitWidth`
Object type name	UpperCamelCase	`RegisterInfo`
Event name	UPPERCASE_WITH_UNDERSCORES	`IRIS_BREAKPOINT_HIT`
Event field name	UPPERCASE_WITH_UNDERSCORES	`BPT_ID`
-	lowercase_with_underscores	Not used.

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 group the functionality further.

See also *4.18.9 Naming conventions for new functions* on page 4-100 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.

4.3 Resources

Resources represent small, named, fixed-sized pieces of state of an instance, for example registers or parameters.

Clients first query a list of all available resources of an instance by calling `resource_getList()`. Optionally, they can query meta information about all resource groups by calling `resource_getListOfResourceGroups()`, but calling this function is not necessary to find or access all resources.

Clients can identify resources by:

- Their name.
- A canonical register id.
- Other aspects, for example tags.

Iris registers are always identified by their `resourceId`, which is an opaque identifier. Resources can be read or written by specifying the `resourceId` in the functions `resource_read()` and `resource_write()`, respectively.

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, see *4.4 Memory* on page 4-59 and *4.6 Tables* on page 4-67.

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 must return an empty list for `resource_getList()`. Each resource has static parts, for example the name and description, and dynamic parts, for example the value.

Parameter and register resources generally behave like normal resources but have extra semantics. For example, parameters can be set at init time. See *4.3.4 Parameters and registers* on page 4-57 for details.

In summary, the Resources interface provides:

- A flat list of named resource groups. The name is a short human-readable string, for example `GPR` or `FPU Shadow`.
- A flat list of named resources. Each resource belongs to one or more resource groups.

This section contains the following subsections:

- *4.3.1 Register fields, sub-registers, and register hierarchy* on page 4-54.
- *4.3.2 Reading a resource* on page 4-55.
- *4.3.3 Writing a resource* on page 4-56.
- *4.3.4 Parameters and registers* on page 4-57.
- *4.3.5 Comparison of resource types* on page 4-57.
- *4.3.6 Exposure of parameters* on page 4-58.

4.3.1 Register fields, sub-registers, and register hierarchy

You can represent register fields, sub-registers, and register hierarchies by using a dot as a hierarchy level separator, for example `CPSR.Z`, or `WPD_MACHINE.STATUS.RUN`.

Registers can be organized into groups. Group names are separated from register names by a dot, for example `MainGroup.CPSR.Z`. The group name is not part of the register name. If a register `foo.bar` exists in group `g`, a register `foo` should also be exposed in group `g`. Clients must handle the case where `foo` does not exist in `g` gracefully, by treating `foo.bar` as a top-level register, without a group. The hierarchy created using this notation should not have more than three levels, that is, with two dots in the name.

Organizing registers into groups and hierarchically within groups is different from organizing registers into a single hierarchy using hierarchical register names. Registers can be in more than one group, and groups are usually presented differently by the client.

As a guideline, a register hierarchy, using the resource name, should be used when registers are physically part of a parent register, while groups should be used when a set of registers belong semantically together.

Clients can display register groups as a linear non-hierarchical list to choose from. Clients should sort this list alphabetically.

4.3.2 Reading a resource

The semantics of reading a resource are *peek* rather than architectural read. All instance implementations should aim for side-effect free reads, which are a prerequisite for non-intrusive debug, although it might not always be possible.

Undefined bits

This interface can indicate which bits in which registers have an undefined value. It is optional for instances to support this.

Approximate values

It is not always possible to return the exact value of a resource. For example, this is the case when the instance that exposes the resource is not in a debuggable state. The `resource_read()` function supports reporting approximate values.

Read errors

The `resource_read()` function does not fail with an error for existing targets and resources. Read errors are indicated in the `error` member of `ResourceReadResult`, for each resource.

Examples

`resource_read()` returns one `ResourceReadResult` object, which contains the read result of all resources 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_ffeeddcc_bbaa9988_77665544_33221100`:

```
ResourceReadResult.data = [0x7766554433221100, 0xffeeddccbbaa9988, 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]
```

Related information

resource_read()

ResourceReadResult

4.3.3 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.

Writing read-only

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 *4.3.2 Reading a resource* on page 4-55. 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.

4.3.4 Parameters and registers

Parameters and registers are resources with additional semantics. A resource cannot be both a parameter and a register.

`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 not implemented and are only stubs. The description of stub registers should indicate that these are only stubs.

Related information

ResourceInfo

4.3.5 Comparison of resource types

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

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

Table 4-2 Comparison of resource types

Criterion	Generic resource	Initialization-time parameter	Run-time parameter	Register
Value is configurable at initialization time.	no	yes	yes	no
Value is modifiable at runtime.	yes	no	yes	yes
Value is modifiable at runtime and generally does not change at runtime except when the user changes it.	yes	no	yes	usually no
Value can change spontaneously, that is, without a parameter write, during runtime.	yes	no	no	yes
Is an architectural register.	no	no	no	yes
Register that has a reset value.	no	no	no	yes
Register that is not reset during simulation reset.	no	yes	yes	no
Corresponds to a design-time parameter of the hardware.	no	yes	no	no
Corresponds to a hardware register.	no	no	no	yes

Table 4-2 Comparison of resource types (continued)

Criterion	Generic resource	Initialization-time parameter	Run-time parameter	Register
Can be modified by program code.	usually no	no	no	yes
Is artificial, not present in the hardware.	yes	yes	yes	usually no

4.3.6 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()
Allows setting 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()
Allows setting initialization-time parameters and run-time parameters. Does not allow setting non-parameter resources.

Initialization-time parameters
Exposed by `resource_getList()`, `resource_read()`, `resource_write()`, `simulation_getInstantiationParameterInfo()`, and `simulation_setInstantiationParameterValues()`. `resource_write()` causes an error.

Run-time parameters
Exposed by `resource_getList()`, `resource_read()`, `resource_write()`, `simulation_getInstantiationParameterInfo()`, and `simulation_setInstantiationParameterValues()`.

Registers
Exposed by `resource_getList()`, `resource_read()`, `resource_write()`.

Generic resources
Exposed by `resource_getList()`, `resource_read()`, `resource_write()`.

Related information

Resources functions
Resources objects

4.4 Memory

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()` functions, respectively. The other memory functions provide more exotic functionality like address translations and retrieving sideband information.

This section contains the following subsections:

- *4.4.1 Accesses* on page 4-59.
- *4.4.2 Errors* on page 4-59.
- *4.4.3 Endianness* on page 4-60.
- *4.4.4 Memory spaces* on page 4-60.
- *4.4.5 Side effects* on page 4-60.
- *4.4.6 Canonical memory space number scheme* on page 4-60.
- *4.4.7 Memory access attributes* on page 4-62.
- *4.4.8 Reading and writing memory* on page 4-63.
- *4.4.9 Reading and writing through caches and buffers* on page 4-63.
- *4.4.10 Address translation* on page 4-64.
- *4.4.11 Memory sideband information* on page 4-64.

4.4.1 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 errors 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 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.

4.4.2 Errors

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

Such errors do not cause the `memory_read()` or `memory_write()` function to return an error status, which would prevent them from returning meaningful return values, 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 per 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.

4.4.3 Endianness

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

4.4.4 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.

4.4.5 Side effects

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

- Component creators should ensure that memory accesses are side-effect free, although it might not always be possible. 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 side-effect free reads.
- 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()`. See `resource_write()` for a description of the side effect behavior.
- 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 hit 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 can never cause a cache inconsistency, but they can remove cache inconsistencies.

4.4.6 Canonical memory space number scheme

The canonical memory space number scheme `arm.com/memoryspaces` is implemented by all Arm components.

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.

The following ids are defined for the `canonicalMsn` member of `MemorySpaceInfo`:

Table 4-3 Canonical memory space number scheme arm.com/memoryspaces

canonicalMsn (NumberU64)	Architecture and configuration	Name	Semantics and translation regime
`0x1000`	Armv8 EL3=AArch64	Secure Monitor	Virtual memory as seen by code running at EL3. This is always secure. This is virtual memory as configured by TCR_EL3.
	Armv7, Armv8, EL3=AArch32	Secure Monitor	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).
	Armv6, Armv7	Secure	Virtual memory as configured by TTBCR(secure).

Table 4-3 Canonical memory space number scheme arm.com/memoryspaces (continued)

canonicalMsn (NumberU64)	Architecture and configuration	Name	Semantics and translation regime
0x1001	Armv8 EL3=AArch64	Guest	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). **Note** Although this is the non-secure bank of TTBCR, this does not make accesses non-secure in this configuration.
	Armv7, Armv8, EL3=AArch32	Guest	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).
	Armv6, Armv7	Normal	This is virtual memory as configured by TTBCR(non-secure).
0x1002	Armv7, Armv8	NS Hyp	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.
0x1003	Armv5, Armv6, Armv7	Memory	Virtual memory. Cores and other components that do not have TrustZone®.
0x1004	Armv7, Armv8	Hyp App	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.
0x1005	Armv8.1	Host	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.
0x10ff	All	Current	Virtual memory view of the current exception level, protection level, or mode. The translation regime used follows the current state of the CPU.
0x1100	Armv7, Armv8	IPA	Intermediate physical memory view. Non-secure.
0x1200	Armv6, Armv7, Armv8	Physical Memory (Secure)	Physical memory, secure world.
0x1201	Armv6, Armv7, Armv8	Physical Memory (Non Secure)	Physical memory, non-secure world.
0x1202	Armv5, Armv6, Armv7	Physical Memory	Physical memory. Cores and components that do not have TrustZone.

Note

- 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.

4.4.7 Memory access attributes

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

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

Note

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

CPU components

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 *4.4.6 Canonical memory space number scheme* on page 4-60 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 4-4 attrib object for Arm CPU components in virtually-addressed regimes

Name	Type	Description
`privileged`	Boolean	Access is privileged.
`instruction`	Boolean	For reads, access is on the instruction side if True, or data side if False.
`user`	NumberU64	User signaling (AXI4).

Table 4-5 attrib object for Arm CPU components in physically-addressed regimes

Name	Type	Description
`nonSecure`	Boolean	Access is non-secure if True, or secure if False.
`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.
`innerCacheability`	String	Cacheability for the inner domain. Must be one of: • `"NC"` (Non-Cacheable). • `"WT"` (Write-Through). • `"WB"` (Write-Back). 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 *4.4.9 Reading and writing through caches and buffers* on page 4-63.

Table 4-5 attrib object for Arm CPU components in physically-addressed regimes (continued)

Name	Type	Description
`outerCacheability`	String	Cacheability for the outer domain. See `innerCacheability`.
`shareability`	String	Shareability. Must be one of: • `"nsh"` (Non-Shareable). • `"ish"` (Inner Shareable). • `"osh"` (Outer Shareable). This is only relevant for `type=Normal` and ignored for other types. **Note** When both `innerCacheability` and `outerCacheability` are `NC`, `shareability` is ignored and is Outer Shareable.

4.4.8 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*.

Read data is transferred in the `data` member of the `MemoryReadResult` object returned by `memory_read()` and write 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. For the format, see the documentation of `memory_write()` or `MemoryReadResult`.

Examples of packing various `byteWidth` elements into `NumberU64` elements:

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
```

byteWidth=8
Reading 1 64-bit word with value 0x8877665544332211:

```
data[0] = 0x8877665544332211
```

byteWidth=16
Reading 1 128-bit word with value 0x1f1e1d1c1b1a19181716151413121110:

```
data[0] = 0x1716151413121110, data[1] = 0x1f1e1d1c1b1a1918
```

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.

4.4.9 Reading and writing through caches and buffers

This information applies to all kinds of buffers, 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.

4.4.10 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`*".

4.4.11 Memory sideband information

Instances can provide sideband information for addresses in a memory space, see `memory_getSidebandInfo()`. GUIs can display this information in a tooltip when the user hovers over a memory cell, for example.

The set of sideband information that is supported depends on the instance and the memory space.

Table 4-6 Sideband info fields

Member	Type	Description
`regionStart`	`NumberU64`	The sideband information in this Object is valid for all addresses in the range `regionStart` to `regionEnd`, 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 `regionStart` or `regionEnd` are missing, the sideband information is only valid for the requested address. The requested address is in the range `regionStart` to `regionEnd`.
`regionEnd`	`NumberU64`	End of the region for which the sideband information is valid. See `regionStart`.

Table 4-6 Sideband info fields (continued)

Member	Type	Description
`physicalAddress`	`NumberU64`	Physical address corresponding to the requested address.
`ipa`	`NumberU64`	Intermediate physical address corresponding to the requested address.
`noExecute`	`Boolean`	If `True`, the requested address cannot be used to execute code.
`ext_<`*`Info`*`>`	`Value`	Components can put arbitrary information here using the `ext_` prefix. Aspects that are supported consistently across multiple components can be added to this table in the future, without the `ext_` prefix.

Related information

Memory functions

Memory objects

4.5 Disassembly

Some instances can provide a disassembled view of their memory. Because disassembly is just a form of `memory_read()`, a memory space id must be provided.

The main interface function is `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 `E_function_not_supported_by_instance` for the `disassembler_*()` functions.

This section contains the following subsections:

- *4.5.1 Disassembling chunks of memory* on page 4-66.
- *4.5.2 Disassembling opcodes* on page 4-66.

4.5.1 Disassembling chunks of memory

Use the `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 `address` field of the last `DisassemblyLine` element of the result value. This function returns `count` lines, unless an error occurred.

This function can return the following errors:

- When `address` is out of the range `minAddr` to `maxAddr` for the memory space, it returns `E_address_out_of_range`.
- When reading past `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 `DisassemblyLine` must be empty and the disassembly string must have the format `"(error: foo)"` where *`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.

4.5.2 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 `disassembler_disassembleOpcode()` function.

Disassembling an opcode never fails because of an invalid opcode. Instead, `opcode=""` and `disass="`*`hex_constant_definition_in_assembler_syntax`*`"` are returned in the `DisassemblyLine` object.

Related information

Disassembly functions

Disassembly objects

4.6 Tables

The tables interface allows an instance to expose an ordered series of records which 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. Enough meta information is provided so that clients can display the information without understanding it.

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.

The following types are allowed in table cells and must be supported by all clients:

- `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.

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.

Related information

Tables functions

Tables objects

4.7 Image loading and saving

This section gives a brief overview of the image loading and saving interface.

This section contains the following subsections:

- *4.7.1 Loading and saving an image* on page 4-68.
- *4.7.2 Image_loadDataRead() callback* on page 4-69.

4.7.1 Loading and saving an image

Use the following functions to load and save images.

Loading an image

`image_loadFile()`
Loads an image into a target instance from a file. Loading an image might cause some state to be modified, for example the start address in the PC register. 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 `image_loadData()` instead, and load the file in the client.

`image_loadData()`
Loads image data into a target instance. The data can come from a file that is accessible to the client instance, for example, but not necessarily accessible to the target instance. `image_loadData()` is semantically equivalent to `image_loadFile()` except that this function does not open a file. Instead, it expects the image contents to be passed as an argument. 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 `image_loadDataPull()`.

`image_loadDataPull()`
Loads image data into a target instance. This is semantically equivalent to `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 `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.

`memory_write()`
This is the primary method of writing raw byte data into memory at a specific address.

Note

- The `image_load*()` functions can optionally load arbitrary raw binary data, which is unformatted and without a header, into memory, by specifying the `rawAddr` and `rawSpaceId` arguments.
- Target instances that do not support all `image_load*()` functions must return `E_function_not_supported_by_instance` for the functions that they do not support.

Saving 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.

In a typical implementation:

- Only target instances that have the `executesSoftware=1` property support the `image_*()` functions.
- Cores and CPUs support the `image_*()` functions.
- Memory components, for example RAMDevice, only support the `memory_*()` functions, not the `image_*()` functions.

4.7.2 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, therefore 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`.

Related information

Image loading and saving functions

Image loading and saving objects

4.8 Simulation time execution control

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.

This section contains the following subsections:

- *4.8.1 Starting and stopping simulation time* on page 4-70.
- *4.8.2 Event source IRIS_SIMULATION_TIME_EVENT* on page 4-70.

4.8.1 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 the simulation time stops progressing or after it stopped 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 is hit, assuming the breakpoint should stop the simulation time.
- An event counter overflow occurs, with `counterMode.overflowStopSim=True`.
- `simulationTime_stop()` is called.
- An instance 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()`.

4.8.2 Event source IRIS_SIMULATION_TIME_EVENT

This event is emitted whenever 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 4-7 Event source IRIS_SIMULATION_TIME_EVENT

Field	Type	Description
`TICKS`	`NumberU64`	Current simulation time in ticks. One tick is 1/`TICK_HZ` seconds long. The elapsed simulation time is `TICKS`/`TICK_HZ` seconds.
`TICK_HZ`	`NumberU64`	Time resolution of the `TICKS` value in Hz. For example, 1000 means that 1 tick = 1ms.
`RUNNING`	`Boolean`	True if and only if the simulation is running, else False. **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 4-7 Event source IRIS_SIMULATION_TIME_EVENT (continued)

Field	Type	Description
REASON	NumberU64	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 *4.9 Debuggable state* on page 4-72. **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.
INST_ID	NumberU64	Optional. If available, this contains the instance id that originally caused the simulation time event.

Related information

Simulation time execution control functions

4.9 Debuggable state

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. These abstract simulations usually 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 approximated values, see `E_approximation`, which is returned in the error field in `ResourceReadResult` and `MemoryReadResult`. It might be unable to provide a value at all, see `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:

- *4.9.1 Debuggable state functions* on page 4-72.
- *4.9.2 debuggableState_setRequest()* on page 4-73.
- *4.9.3 debuggableState_getAcknowledge()* on page 4-73.
- *4.9.4 simulationTime_runUntilDebuggableState()* on page 4-73.

4.9.1 Debuggable state functions

The functions that are defined in this API allow you to bring one or more instances in a system into a debuggable state.

To achieve this, instances that have a concept of debuggable state maintain the following flags:

Debuggable-state-request flag
: This is set and cleared by the client to request that this instance should go into a debuggable state. See *4.9.2 debuggableState_setRequest()* on page 4-73.

Debuggable-state-acknowledge flag
: This is set and cleared by the instance to indicate to clients that this instance is in a debuggable state. See *4.9.3 debuggableState_getAcknowledge()* on page 4-73.

These flags have similar semantics to a debug request pin and a debug acknowledge pin found on some CPUs.

In addition to these two instance-specific flags, there is a global, non-instance-specific function, `simulationTime_runUntilDebuggableState()` which advances simulation time until all instances for which a debuggable state is currently requested, have acknowledged it.

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.

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 `simulationTime_runUntilDebuggableState()` is usually only supported if and only if there are one or more instances in the system that support the `debuggableState_*()` functions.

4.9.2 debuggableState_setRequest()

This function sets or clears the `debuggable-state-request` flag in a specific instance.

Only this function can change the flag. The flag is per-instance and not per-client.

Setting the flag in an instance 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.

4.9.3 debuggableState_getAcknowledge()

This function is used to query the `debuggable-state-acknowledge` flag of an instance.

The instance automatically sets the flag when it reaches a debuggable state. This is typically only the case after setting the `debuggable-state-request` flag and after executing some simulation time, usually by using `simulationTime_runUntilDebuggableState()`.

The instance automatically clears the `debuggable-state-acknowledge` flag when it leaves the debuggable state. This is usually the case when simulation time progresses while the `debuggable-state-request` flag is cleared.

This function has no side effects on the instance.

4.9.4 simulationTime_runUntilDebuggableState()

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 `simulation_runUntilDebuggableState()`, this does not cause the simulation time to stop progressing when all the instances enter a debuggable state.

These are the typical steps to inspect a model that supports debuggable state:

1. Call `debuggableState_setRequest(request = true)` on all instances that should be inspected and that support the `debuggableState_setRequest()` function.
2. Call `simulationTime_runUntilDebuggableState()`.
3. Wait until `IRIS_SIMULATION_TIME_EVENT` occurs.
4. Inspect the `REACHED_DEBUGGABLE_STATE` bit of the `REASON` field of `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 information

Debuggable state functions

Function-specific error codes

4.10 Stepping

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:

- *4.10.1 Step units* on page 4-75.
- *4.10.2 Step counters* on page 4-75.
- *4.10.3 Examples* on page 4-75.

4.10.1 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.

4.10.2 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, or `simulationTime_stop()` was called.

4.10.3 Examples

To step an instance, a client calls `step_setup()` on it to set the *remaining steps* counter to `N` steps. This function does not execute any steps and also 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:

```
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 *4.11 Per-instance execution control* on page 4-77. The following sequence progresses one specific core in a system that contains multiple cores, by `N` steps. This sequence uses per-instance execution control:

```
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 steps have been executed, for example because a breakpoint is hit or `simulationTime_stop()` is 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 information

Stepping functions

4.11 Per-instance execution control

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 when the simulation time of the whole system progresses. Therefore, the per-instance execution control cannot progress the state of individual components while 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.

Related information

Per-instance execution control functions

4.12 Breakpoints

Clients manipulate breakpoints in the instance by using the `breakpoint_set()` and `breakpoint_clear()` 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 invisible and undeletable breakpoints. 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 are specific to the target instance that contains them, not to the client that sets them. Breakpoint ids are specific to the instance.

The target instance implementation must ensure that after a breakpoint is hit, which stops 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.

This section contains the following subsections:

- *4.12.1 Breakpoint actions and trace points* on page 4-78.
- *4.12.2 Event source IRIS_BREAKPOINT_HIT* on page 4-79.
- *4.12.3 Other ways to stop simulation time* on page 4-79.

4.12.1 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 in addition to the `IRIS_BREAKPOINT_HIT` callback and 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 DS-5 Debugger 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.

4.12.2 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 4-8 Event source IRIS_BREAKPOINT_HIT

Field	Type	Description
BPT_ID	NumberU64	Breakpoint id of the breakpoint that was hit.
PC	NumberU64	PC value when the breakpoint was hit.
PC_SPACE_ID	NumberU64	Memory space id of the PC value when the breakpoint was hit.
ACCESS_ADDR	NumberU64	Optional. Address of the access that hit a data breakpoint. Mandatory for data breakpoints. Not present for other breakpoint types.
ACCESS_SIZE	NumberU64	Optional. Size in bytes of the access that hit a data breakpoint. Mandatory for data breakpoints. Not present for other breakpoint types.
ACCESS_RW	String	Optional. Either `"r"` or `"w"`. The `rwMode` of the access that hit a data or register breakpoint. Mandatory for these breakpoint types. Not present for other breakpoint types.
ACCESS_DATA	NumberU64[]	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.
TYPE	String	Breakpoint type. One of: • `code` • `data` • `register` See `breakpoint_set()`.
DONTSTOP	Boolean	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.

4.12.3 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 *4.14.4 Event callback functions* on page 4-85.

Stepping

The stepping functionality stops the simulation time after a specified number of steps, see *4.10 Stepping* on page 4-75.

Debuggable state

The debuggable state functionality allows clients to stop simulation time when one or more instances reaches a debuggable state, see *4.9 Debuggable state* on page 4-72.

Related information

Breakpoints functions

Breakpoints objects

4.13 Notification and discovery of state changes

Clients observe the state of component instances and might display this state to the user. Clients can actively query this state from the components, so they need to know when the state is updated.

While the simulation time is progressing, the state of all component instances might change spontaneously and continuously. No special notifications are sent about any state changes, although some instances support event sources to inform clients about events.

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 a component instance, for example by writing resources or memory. Other clients should register for the `IRIS_STATE_CHANGED` event to get notification of these changes, to update their views. When a client calls a state-modifying function like `resource_write()` or `memory_write()` the `IRIS_STATE_CHANGED` event is automatically generated. The client does not need to generate it. See *4.13.1 Event source IRIS_STATE_CHANGED* on page 4-81.

4.13.1 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 `resource_write()`.
- Memory contents. See `memory_write()` and `image_load*()`.
- Table contents. See `table_write()`.
- Sync level. See `syncLevel_request()` and `syncLevel_release()`.

This event is not guaranteed to be generated after changes to the following:

- List of registered instances. See `IRIS_INSTANCE_REGISTRY_CHANGED` for this change.
- Active event streams. This should be transparent to other instances, by definition.
- Run or stop state of the simulation time. See `IRIS_SIMULATION_TIME_EVENT` 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 suppress multiple events from occurring because of many state changes 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.

4.14 Events and trace interface

All Iris events are handled through this 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.

Optionally, events can be enabled or disabled 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. All other event functions are exotic and deal with ringbuffering events, counter events, and reading the state of an event.

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()`.

Target 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 these events, and allow clients to receive them. In particular, instances that support the breakpoint interface or the semihosting interface must implement the event interface.

This section contains the following subsections:
- *4.14.1 Ring buffers* on page 4-82.
- *4.14.2 References in event source fields* on page 4-83.
- *4.14.3 Creating and destroying event streams* on page 4-84.
- *4.14.4 Event callback functions* on page 4-85.
- *4.14.5 Event counters* on page 4-85.

4.14.1 Ring buffers

Events can optionally be stored in a ring buffer on the simulation side. One reason to do this is to improve runtime performance.

The buffer can also be used if only the most recent events are of interest to the client. This feature could be implemented on the client side using the normal `ec_FOO()` callbacks but the performance would be slow.

Each client has its own ring buffer on the simulation side. Each ring buffer can buffer events from all instances. The effect is that each client can control its ring buffer as if it was implemented on the client side, with the performance advantages of implementing it on the simulation side. Clients can freely use their own ring buffer without affecting other clients.

Ring buffers support the following modes, see the `mode` argument of `eventRingBuffer_init()`:

`mode="overwrite"`
Continuously overwrite old data.

`mode="send"`
Send buffered events to clients in chunks.

`mode="drop"`
Fill up only once until full.

Clients can configure the size of their ring buffer, read from it, and clear it.

To keep the internal event infrastructure simple and fast, each client only has one ring buffer. Clients can easily demultiplex the event streams that go into the buffer, so this limitation is rarely a problem. If a client needs multiple ring buffers, it can register extra client instances, each of which has its own ring buffer. This is rarely necessary. A possible use case is to have a large buffer capturing only a PC trace and a smaller buffer capturing the last ten memory transactions, each containing many fields, which would quickly fill up the PC trace buffer.

If a ring buffer is not used for a particular event stream, which is the default, events are sent to the client by the `ec_FOO()` callback, synchronously or asynchronously, as they occur.

4.14.2 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 include a field that identifies a resource must use the numeric resource id as the field, see `ResourceInfo.rscId`, not the resource name string. If resource names are reported, which is discouraged, they must be consistent with the `ResourceInfo.name` field.

Event sources that include a field that identifies a memory space must use the `spaceId` as the field, see `MemorySpaceInfo.spaceId`. The same restrictions apply when referring to memory spaces as to resources.

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 use cases:

`%{<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 to 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 within 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.

4.14.3 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 an explicit or implicit `instanceRegistry_unregisterInstance()`.

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 mechanisms 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 *4.16 Simulation accuracy (sync levels)* on page 4-92.

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 on 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 information

eventStream_setTraceRanges()

4.14.4 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.

4.14.5 Event counters

To count events in an event stream without the runtime overhead of using the 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 each 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 relevant 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 be able to determine the number of wraparounds that occurred.

Related information

Events and trace interface functions

Events and trace interface objects

4.15 Semihosting

Iris provides basic support for stdin, stdout, and stderr. It also supports the addition of new semihosting functions and replacement of existing semihosting implementations.

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.

This section contains the following subsections:

- *4.15.1 Basic stdin, stdout, and stderr support* on page 4-87.
- *4.15.2 Event source IRIS_SEMIHOSTING_OUTPUT* on page 4-88.
- *4.15.3 Semihosting input interface* on page 4-88.
- *4.15.4 Event source IRIS_SEMIHOSTING_INPUT_REQUEST* on page 4-89.
- *4.15.5 Event source IRIS_SEMIHOSTING_INPUT_UNBLOCKED* on page 4-89.
- *4.15.6 Event source IRIS_SEMIHOSTING_CALL* on page 4-89.
- *4.15.7 Extending or replacing the semihosting implementation* on page 4-90.
- *4.15.8 Event source IRIS_SEMIHOSTING_CALL_EXTENSION* on page 4-90.

4.15.1 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 provided as an event source, `IRIS_SEMIHOSTING_OUTPUT`, which is defined for components that support semihosting output on this abstraction level. 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. Their 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.

4.15.2 Event source IRIS_SEMIHOSTING_OUTPUT

This event is generated by the target instance to emit bytes of semihosting output.

Table 4-9 Event source IRIS_SEMIHOSTING_OUTPUT

Field	Type	Description
DATA	NumberU64[]	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.
SIZE	NumberU64	Number of bytes in DATA.
FDES	NumberU64	File descriptor number that is used for output. 1 means stdout and 2 means stderr. Other values have target instance or target application-specific semantics.

4.15.3 Semihosting input interface

Semihosting input is implemented using event sources and functions.

Enabling semihosting input typically involves the following steps:

- The client activates the IRIS_SEMIHOSTING_INPUT_REQUEST and, optionally, IRIS_SEMIHOSTING_INPUT_UNBLOCKED event sources 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, and also they all might provide input that is interleaved and unsynchronized.
- Applications that do not require input do not issue an IRIS_SEMIHOSTING_INPUT_REQUEST event.
- When the application requires user input, it causes the semihosting implementation to issue an IRIS_SEMIHOSTING_INPUT_REQUEST event. The client might read from the console or open a terminal window UI, for example.
- When the user has entered either one character or one line of data, the client passes this data to the semihosting interface using the function semihosting_provideInputData().
- If the simulated application wants more data, the client waits for user input and sends it to the semihosting interface when it is available. If the application does not want 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.
- The client can send user input to the semihosting implementation using semihosting_provideInputData() at any time, even if the running application is not waiting for data. This can happen 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_REQEST must assume that the IRIS_SEMIHOSTING_INPUT_REQUEST RAW field is False. This means user input should be fed into the model when the user presses the Enter key, in other words terminal *cooked* mode.

  ———— **Note** ————

  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 into the model without the model processing it. The client should send a maximum of 1MB of data to the simulation before receiving an IRIS_SEMIHOSTING_INPUT_REQUEST event. After receiving an IRIS_SEMIHOSTING_INPUT_REQUEST event, the client can send a maximum of SIZEHINT + 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.

4.15.4 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 4-10 Event source IRIS_SEMIHOSTING_INPUT_REQUEST

Field	Type	Description
FDES	NumberU64	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.
NONBLOCK	Boolean	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.
RAW	Boolean	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.
SIZEHINT	NumberU64	Number of bytes that the model consumes immediately.

4.15.5 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 because they never block. The raw or cooked mode should be set to cooked, see `IRIS_SEMIHOSTING_INPUT_REQUEST RAW` field.

Table 4-11 Event source IRIS_SEMIHOSTING_INPUT_UNBLOCKED

Field	Type	Description
FDES	NumberU64	File descriptor number used for input. This is usually 0, which means stdin.

4.15.6 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 4-12 Event source IRIS_SEMIHOSTING_CALL

Field	Type	Description
OPERATION	NumberU64	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.
PARAMETER	NumberU64	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.

4.15.7 Extending or replacing the semihosting implementation

Targets can contain a semihosting implementation that provides libc functionality. Applications can use this libc 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 return value 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.

4.15.8 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 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 4-13 Event source IRIS_SEMIHOSTING_CALL_EXTENSION

Field	Type	Description
`OPERATION`	`NumberU64`	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.
`PARAMETER`	`NumberU64`	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.

Related information

Semihosting functions

4.16 Simulation accuracy (sync levels)

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 subsection:

- *4.16.1 Sync points* on page 4-92.

4.16.1 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 4-14 Sync levels and sync points

Sync level	Description
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 that are used for software development. • Normal debugging sessions when no watchpoint is set. The sync point 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 (`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. Use cases: • External breakpoints that block the simulation. • Inspecting the processor state from within peripheral accesses.

Table 4-14 Sync levels and sync points (continued)

Sync level	Description
2	`POST_INSN_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 completed, post-instruction. Use cases: • Watchpoints. • External breakpoints (breakpoints in peripherals). • 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	`POST_INSN_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 completed (post-instruction). 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.

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, in other words, breakpoints that are set in peripherals, behind a bridge, 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 information

Simulation accuracy (sync level) functions

Simulation accuracy (sync level) objects

4.17 Checkpointing

This section describes the checkpointing functions that Iris supports.

This section contains the following subsection:

- *4.17.1 simulation_reset()* on page 4-94.

4.17.1 simulation_reset()

This global function resets the simulation to the same state it had after it was instantiated with `simulation_instantiate()`.

If simulation time is progressing, the simulation is stopped before resetting the system. The simulation time is not resumed after the reset is finished.

A hardware reset can leave some state undefined. A simulation reset must leave all state defined, even state that is architecturally marked as *undefined after reset*.

A simulation reset must also reset all internal state machines, for example quantum counters.

Related information

simulation_reset()

4.18 Instance registry, instance discovery, and interface discovery

All entities in Iris are represented by instances. Instances must be registered in the global instance registry. Instances can discover, communicate with, and check which functions are supported by other instances.

This section contains the following subsections:

- *4.18.1 Hierarchical instance names and instance classes* on page 4-95.
- *4.18.2 Registering instances* on page 4-96.
- *4.18.3 Unregistering instances* on page 4-96.
- *4.18.4 Event source IRIS_INSTANCE_REGISTRY_CHANGED* on page 4-97.
- *4.18.5 Instance and target information* on page 4-98.
- *4.18.6 Interface discovery* on page 4-99.
- *4.18.7 instance_ping()* on page 4-100.
- *4.18.8 Interface versioning* on page 4-100.
- *4.18.9 Naming conventions for new functions* on page 4-100.

4.18.1 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 a 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. The instance class does not limit what the

instance can do, for example producing or consuming events. It only determines in which list it appears to other instances when enumerating instances.
- 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.

4.18.2 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 determine where to route requests and responses that are targeted at `foo.bar`.

`instanceRegistry_registerInstance()` must be called as a request, not as a notification. When it is called as a notification, this call is ignored and no instance is registered.

Instances must call this function before calling any other Iris function because their instance id must be included in any request ids. When calling this function, instances do not yet have an instance id assigned, so they cannot fill their instance id into 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 hierarchy 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 *4.18.3 Unregistering instances* on page 4-96 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.

4.18.3 Unregistering instances

`instanceRegistry_unregisterInstance()` unregisters an instance from the instance registry. It undoes all the effects of `instanceRegistry_registerInstance()`.

When this function returns, the instance is no longer visible in `instanceRegistry_getList()`. Other 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 of making 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 ongoing 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 can 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 `instId` that they no longer know about 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. It must assume that the `instId` instance no longer exists.

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.

4.18.4 Event source IRIS_INSTANCE_REGISTRY_CHANGED

This event is generated whenever the list of registered instances has changed. It is generated shortly after the change was applied to the instance registry.

This event source is only provided by `framework.GlobalInstance`, not by individual instances.

Table 4-15 Event source IRIS_INSTANCE_REGISTRY_CHANGED

Field	Type	Description
EVENT	String	The event that occurred. Possible values: "added" Added a new instance to the instance registry. "removed" Removed an instance from the instance registry.
INST_ID	NumberU64	Instance id of the instance that was added or removed.
INST_NAME	String	Instance name of the instance that was added or removed.

4.18.5 Instance and target information

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 4-16 Instance properties defined by Iris, typically only for component instances

Property	Type	Description
register.canonicalRnScheme	String	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.
memory.canonicalMsnScheme	String	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/memoryspaces, if they expose memory spaces, see *4.4.6 Canonical memory space number scheme* on page 4-60.

Table 4-17 Instance properties defined by LISA+, typically only for component instances

Property	Type	Description
componentName	String	The name of the component that this is an instance of. For example RAMDevice.
version	String	Component version, as defined in the LISA+ properties section.

Table 4-17 Instance properties defined by LISA+, typically only for component instances (continued)

Property	Type	Description
componentType	String	Component type. The following types are defined: • Bus • Core • Cluster • Peripheral • System • Other 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.
description	String	Short description of the functionality of the component. Can contain linefeeds, but is usually just a single line of text.
documentationFile	String	Filename of the documentation for this component. A user interface can open this file upon request. The LISA+ name is documentation_file.
executesSoftware	NumberU64	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.
loadfileExtension	String	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 "*.txt" or "*.axf;*.elf". The LISA+ name is loadfile_extension.

Table 4-18 Instance properties for client instances and for instances connected using IPC

Property	Type	Description
name	String	Human readable name of a client. Single line, usually prettier than the instance name, for example "Arm Development Studio v1.0". This does not need to be unique for different clients, if they are of the same type.
description	String	Multiple-line description of this client.
connectionInfo	String	Single line that describes the way this instance is connected to the simulation. This can be either: **"in-process"** For DSO plug-ins, for example. This should be the default interpretation when this property is missing. **"iris://<*ip*>:<*port*>"** For an IPC client on <*ip*> and <*port*>.

4.18.6 Interface discovery

Instances can support any subset of Iris functions. It is possible to check whether a function or a set of functions is supported by a specific instance. It is also possible to get a list of all functions that are supported by a specific instance.

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`.

4.18.7 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.

4.18.8 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:

- A caller can use `instance_checkFunctionSupport()` to determine whether a specific function or set of functions is supported. It 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 in the following ways:
  - If new details are returned for certain queries, for example static register information, the return value objects are extended with new members. Existing clients ignore these additions and new clients can reliably see them. The target is extended without breaking existing clients.
  - 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 do not specify these new arguments and receive the previous behavior. The target is extended without breaking existing clients.
  - 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 do not know about this new function and 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.

4.18.9 Naming conventions for new functions

When adding new functions that replace or extend existing functions, use the following naming conventions.

Note

- Only introduce 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 *4.2 Naming conventions* on page 4-53 for general naming conventions.

Related information

Instance registry, instance discovery, and interface discovery functions
Instance registry, instance discovery, and interface discovery objects

4.19 Simulation instantiation and discovery

This section describes the functionality to instantiate simulations and discover running simulations.

There are two main use cases when using simulations:

- Connecting to an already 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 inside 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:

- *4.19.1 Connecting to a running simulation using IPC* on page 4-102.
- *4.19.2 Event source IRIS_SIMULATION_SHUTDOWN_ENTER* on page 4-102.
- *4.19.3 Event source IRIS_SIMULATION_SHUTDOWN_LEAVE* on page 4-103.
- *4.19.4 Instantiating a new simulation, in-process* on page 4-103.
- *4.19.5 Instantiation parameters* on page 4-103.
- *4.19.6 Setting instantiation parameter values* on page 4-104.

4.19.1 Connecting to a running simulation using IPC

Connecting to a running simulation typically follows this sequence:

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.

4.19.2 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.

4.19.3 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.

4.19.4 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.

4.19.5 Instantiation parameters

The parameters that are exposed through `simulation_getInstantiationParameterInfo()` are called *instantiation parameters*.

Instantiation parameter values 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 refer to parameters that 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 refer to parameters that 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.

4.19.6 Setting instantiation parameter values

Pre-instantiation clients can call `simulation_setInstantiationParameterValues()` to initialize all init-time and runtime 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. It can be called multiple times. Each invocation can set all of the instantiation parameters, or a subset of them. Each invocation can overwrite parameter values that were set with previous invocations of `simulation_setInstantiationParameterValues()`.

Calling this function is optional. Any parameters that this function does not set keep their default values, which are used to instantiate the system.

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()`.

Related information

Simulation instantiation and discovery functions

Simulation instantiation and discovery objects

4.20 Plug-in loading and instantiation

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, but the factory instance remains, allowing multiple instances of the plug-in to be created 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.

Related information

Plugins loading and instantiation functions

4.21 Iris-text-format

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:

- *4.21.1 Format strings* on page 4-106.
- *4.21.2 References to variables* on page 4-106.
- *4.21.3 Conditional formatting* on page 4-108.

4.21.1 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.

4.21.2 References to variables

References to variables generally have the following syntax:

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

Note

The percentage sign, braces {…}, and the square brackets […] around *optional_bitrange* 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-x|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.
```

4.21.3 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]]
```

4.22 instId argument

An `instId` argument occurs in many different functions, and has the same meaning in all of them.

The `instId` argument defines the instance that a function call is sent to. It plays 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:

```
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, even 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.

4.23 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 without breaking compatibility. To achieve this compatibility, 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 the Iris documentation.
- 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 the Iris documentation.
- 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 the Iris documentation.
- 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 when enhancing interfaces:

- Callees can be enhanced to accept more optional arguments.
- Callees can be enhanced so that they return additional return value members. Existing callers that are unaware of new members 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.

4.24 TCP server management

The global instance manages the Iris TCP server.

The TCP server can be started or stopped at any phase 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.

Related information

TCP server functions