IrisSupportLib

Version 1.0

Reference Guide

Chapter 1

IrisSupportLib Reference Guide

About this document

This book contains API reference documentation for IrisSupportLib. It was generated from the source code using Doxygen.

The IrisSupportLib library contains the code to create an IrisInstance object and helper classes to add functionality to the instance. It also contains the code to communicate with the Iris system using U64JSON and general support code used by the library, for example thread abstraction.

IrisSupportLib is built as a static library. It must be linked in to any executable or DSO that needs to connect to Iris. The library is provided pre-compiled in `$IRIS_HOME/<OS_Compiler>/libIrisSupport.a` or `IrisSupport.lib`. Headers are provided in the directory `$IRIS_HOME/include/iris/` and the source code is provided in the directory `$IRIS_HOME/IrisSupportLib/`.

Useful resources

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

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available on Arm Developer. Each document link in the following lists goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm® product resources

For more information about Iris, see the Iris User Guide.
Note Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader. Adobe PDF reader products can be downloaded at http://www.adobe.com.

See the following Iris client and plug-in examples:

- `$IRIS_HOME/Examples/Client/` for Iris C++ client examples.
- `$IRIS_HOME/Python/Examples/` for Iris Python client examples.
- `$IRIS_HOME/Examples/Plugin/` for Iris plug-in examples.

Release information

Table 1.1 Document history

Issue	Date	Confidentiality	Change
0100-00	23 Nov 2018	Non-Confidential	New document for Fast Models v11.5.
0100-01	26 Feb 2019	Non-Confidential	Update for v11.6.
0100-02	17 May 2019	Non-Confidential	Update for v11.7.
0100-03	05 Sep 2019	Non-Confidential	Update for v11.8.
0100-04	28 Nov 2019	Non-Confidential	Update for v11.9.
0100-05	12 Mar 2020	Non-Confidential	Update for v11.10.
0100-06	22 Sep 2020	Non-Confidential	Update for v11.12.
0100-07	09 Dec 2020	Non-Confidential	Update for v11.13.
0100-08	17 Mar 2021	Non-Confidential	Update for v11.14.
0100-09	29 Jun 2021	Non-Confidential	Update for v11.15.
0100-10	06 Oct 2021	Non-Confidential	Update for v11.16.
0100-11	16 Feb 2022	Non-Confidential	Update for v11.17.
0100-12	15 Jun 2022	Non-Confidential	Update for v11.18.
0100-13	14 Sept 2022	Non-Confidential	Update for v11.19.
0100-14	07 Dec 2022	Non-Confidential	Update for v11.20.
0100-15	22 Mar 2023	Non-Confidential	Update for v11.21.
0100-16	14 Jun 2023	Non-Confidential	Update for v11.22.
0100-17	13 Sep 2023	Non-Confidential	Update for v11.23.
0100-18	06 Dec 2023	Non-Confidential	Update for v11.24.
0100-19	13 Mar 2024	Non-Confidential	Update for v11.25.
0100-20	19 Jun 2024	Non-Confidential	Update for v11.26.

Proprietary Notice

to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

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

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

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

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

The validity, construction and performance of this notice shall be governed by English Law.
The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. Please follow Arm's trademark usage guidelines at https://www.arm.com/company/policies/trademarks. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners.

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.
PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential.

Product Status

All products and Services provided by Arm require deliverables to be prepared and made available at different levels of completeness. The information in this document indicates the appropriate level of completeness for the associated deliverables.

Product completeness status

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

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on Arm Support.

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

Inclusive language commitment

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

This document includes language that can be offensive. We will replace this language in a future issue of this document.
To report offensive language in this document, email terms@arm.com.

Chapter 2

IrisSupportLib NAMESPACE macros

To allow multiple different versions of IrisSupportLib to be used by different components in the same executable, all IrisSupportLib code is defined in a hidden inner namespace. This namespace is constructed from the revision and fork from `iris/detail/IrisSupportLibRevision.h`. For example, if revision=0 and fork=master, this means IrisSupportLib code is in the namespace `iris::r0master`.
This is then imported into the namespace `iris` so all Iris code can be used without the hidden internal namespace. Make sure you include the Iris `NAMESPACE_` macros in any new source files, for example:

```
...
#ifndef ARM_INCLUDE_MyHeader_h
#define ARM_INCLUDE_MyHeader_h

#include "iris/detail/IrisCommon.h"

NAMESPACE_IRIS_START

// Code goes here

NAMESPACE_IRIS_END

#endif // ARM_INCLUDE_MyHeader_h
```

Chapter 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Hierarchical Index

4.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 5

Class Index

5.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 6

File Index

6.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 7

Module Documentation

7.1 Instance Flags

Flags that can be set when registering an IrisInstance.

Variables

- static const bool **iris::IrisInstance::ASYNCHRONOUS** = !SYNCHRONOUS

  Cause enableEvent() callback to be called back asynchronously (i.e. the caller does not wait for the function call to return).
- static const uint64_t **iris::IrisInstance::DEFAULT_FLAGS** = THROW_ON_ERROR

  Default flags used if not otherwise specified.
- static const bool **iris::IrisInstance::SYNCHRONOUS** = true

  Cause enableEvent() callback to be called back synchronously (i.e. the caller is blocked until the callback function returns).
- static const uint64_t **iris::IrisInstance::THROW_ON_ERROR** = (1 << 1)

  Throw an exception when an Iris call returns an error response.
- static const uint64_t **iris::IrisInstance::UNIQUIFY** = (1 << 0)

  Uniquify instance name when registering.

7.1.1 Detailed Description

Flags that can be set when registering an IrisInstance.

7.2 IrisInstanceBuilder resource APIs

Set up resource and register metadata and delegates.

Classes

- class iris::IrisInstanceBuilder::FieldBuilder

  Used to set metadata on a register field resource.
- class iris::IrisInstanceBuilder::ParameterBuilder

  Used to set metadata on a parameter.
- class iris::IrisInstanceBuilder::RegisterBuilder

  Used to set metadata on a register resource.

Functions

- RegisterBuilder iris::IrisInstanceBuilder::addNoValueRegister (const std::string &name, const std::string &description, const std::string &format)

  Add metadata for one noValue resource.
- ParameterBuilder iris::IrisInstanceBuilder::addParameter (const std::string &name, uint64_t bitWidth, const std::string &description)

  Add numeric parameter.
- RegisterBuilder iris::IrisInstanceBuilder::addRegister (const std::string &name, uint64_t bitWidth, const std::string &description, uint64_t addressOffset=IRIS_UINT64_MAX, uint64_t canonicalRn=IRIS_UINT64_MAX)

  Add metadata for one numeric register resource.
- ParameterBuilder iris::IrisInstanceBuilder::addStringParameter (const std::string &name, const std::string &description)

  Add string parameter.
- RegisterBuilder iris::IrisInstanceBuilder::addStringRegister (const std::string &name, const std::string &description)

  Add metadata for one string register resource.
- void iris::IrisInstanceBuilder::beginResourceGroup (const std::string &name, const std::string &description, uint64_t subRscIdStart=IRIS_UINT64_MAX, const std::string &cname=std::string())

  Begin a new resource group.
- ParameterBuilder iris::IrisInstanceBuilder::enhanceParameter (ResourceId rscId)

  Get ParameterBuilder to enhance a parameter.
- RegisterBuilder iris::IrisInstanceBuilder::enhanceRegister (ResourceId rscId)

  Get RegisterBuilder to enhance register.
- const ResourceInfo & iris::IrisInstanceBuilder::getResourceInfo (ResourceId rscId)

  Get ResourceInfo of a previously added register.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) READER, IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &) WRITER>
  void iris::IrisInstanceBuilder::setDefaultResourceDelegates (T *instance)

  Set both read and write resource delegates if they are defined in the same class.
- template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
  void iris::IrisInstanceBuilder::setDefaultResourceReadDelegate ()

  Set default read access function for all subsequently added resources.
- void iris::IrisInstanceBuilder::setDefaultResourceReadDelegate (ResourceReadDelegate delegate=ResourceReadDelegate())

  Set default read access function for all subsequently added resources.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
  void iris::IrisInstanceBuilder::setDefaultResourceReadDelegate (T *instance)

  Set default read access function for all subsequently added resources.
- template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
  void iris::IrisInstanceBuilder::setDefaultResourceWriteDelegate ()

  Set default write access function for all subsequently added resources.
- void iris::IrisInstanceBuilder::setDefaultResourceWriteDelegate (ResourceWriteDelegate delegate=ResourceWriteDelegate())

  Set default write access function for all subsequently added resources.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &) METHOD>
  void iris::IrisInstanceBuilder::setDefaultResourceWriteDelegate (T *instance)

  Set default write access function for all subsequently added resources.
- void iris::IrisInstanceBuilder::setNextSubRscId (uint64_t nextSubRscId)

  Set the rscId that will be used for the next resource to be added.
- void iris::IrisInstanceBuilder::setPropertyCanonicalRnScheme (const std::string &canonicalRnScheme)

  Set the register.canonicalRnScheme instance property.
- void iris::IrisInstanceBuilder::setTag (ResourceId rscId, const std::string &tag)

  Set a tag for a specific resource.

7.2.1 Detailed Description

Set up resource and register metadata and delegates.

7.2.2 Function Documentation

7.2.2.1 addNoValueRegister()

```
RegisterBuilder iris::IrisInstanceBuilder::addNoValueRegister (
            const std::string & name,
            const std::string & description,
            const std::string & format )
```

Add metadata for one noValue resource.
Resource group: beginResourceGroup() must have been called before calling this function. The added resource is automatically added to the last group added by beginResourceGroup().
Type: The added resource is of type 'noValue'. Use addRegister() to add a register of type 'numeric' or 'numericFp'. Use addStringRegister() to add a register of type 'string'.
The returned builder object is only valid until another resource is added. It is only intended to modify the resource that was added last.

Parameters

name	Name of the resource. This is the same as the 'name' field of ResourceInfo.
description	Human readable description of the resource. This is the same as the 'description' field of ResourceInfo.
format	The format used to display this resource.

Returns

A RegisterBuilder object that can be used to set additional metadata for this resource.

7.2.2.2 addParameter()

```
ParameterBuilder iris::IrisInstanceBuilder::addParameter (
            const std::string & name,
            uint64_t bitWidth,
            const std::string & description )
```

Add numeric parameter.
Resource group: beginResourceGroup() must have been called before calling this function. The added parameter is automatically added to the last group added by beginResourceGroup().
Type: The added parameter is of type 'numeric'. Call setType("numericFp") on the returned ParameterBuilder to add a 'numericFp' (pure floating point) parameter. Use addStringParameter() to add a parameter of type 'string'.
The returned builder object is only valid until another resource is added. It is only intended to modify the resource that was added last.

Parameters

name	Name of the parameter. This is the same as the 'name' field of ResourceInfo.
bitWidth	Width of the parameter in bits. This is the same as the 'bitWidth' field of ResourceInfo.
description	Human readable description of the parameter. This is the same as the 'description' field of ResourceInfo.

Returns

A ParameterBuilder object that can be used to set additional metadata for this parameter.

7.2.2.3 addRegister()

```
RegisterBuilder iris::IrisInstanceBuilder::addRegister (
            const std::string & name,
            uint64_t bitWidth,
            const std::string & description,
            uint64_t addressOffset = IRIS_UINT64_MAX,
            uint64_t canonicalRn = IRIS_UINT64_MAX )
```

Add metadata for one numeric register resource.
Resource group: beginResourceGroup() must have been called before calling this function. The added resource is automatically added to the last group added by beginResourceGroup().
Type: The added resource is of type 'numeric'. Call setType("numericFp") on the returned RegisterBuilder to add a 'numericFp' (pure floating-point) register. Use addStringRegister() to add a register of type 'string'.
The returned builder object is only valid until another resource is added. It is only intended to modify the resource that was added last.

Parameters

name	Name of the register. This is the same as the 'name' field of ResourceInfo.
bitWidth	Width of the resource in bits. This is the same as the 'bitWidth' field of ResourceInfo.
description	Human readable description of the resource. This is the same as the 'description' field of ResourceInfo.
addressOffset	The address offset of this register inside the parent device. This is the same as the 'addressOffset' field of RegisterInfo.
canonicalRn	Canonical Register Number. This is the same as the 'canonicalRn' field of RegisterInfo.

Returns

A RegisterBuilder object that can be used to set additional metadata for this register resource.

Remarks

A value of 2**64-1 (0xFFFFFFFFFFFFFFFF) for the arguments *addressOffset* and *canonicalRn* (the default value) is used to indicate that the field is not set. To set an addressOffset of 2**64-1 use
`addRegister(...).setAddressOffset(iris::IRIS_UINT64_MAX);`

To set a caconicalRn of 2**64-1 use
`addRegister(...).setCanonicalRn(iris::IRIS_UINT64_MAX);`

7.2.2.4 addStringParameter()

```
ParameterBuilder iris::IrisInstanceBuilder::addStringParameter (
            const std::string & name,
            const std::string & description )
```

Add string parameter.
Resource group: beginResourceGroup() must have been called before calling this function. The added parameter is automatically added to the last group added by beginResourceGroup().
Type: The added parameter is of type 'string'. Use addParameter() to add a parameter of a type 'numeric' or 'numericFp'.
The returned builder object is only valid until another resource is added. It is only intended to modify the resource that was added last.

Parameters

name	Name of the parameter. This is the same as the 'name' field of ResourceInfo.
description	Human readable description of the parameter. This is the same as the 'description' field of ResourceInfo.

Returns

A ParameterBuilder object that can be used to set additional metadata for this parameter.

7.2.2.5 addStringRegister()

```
RegisterBuilder iris::IrisInstanceBuilder::addStringRegister (
            const std::string & name,
            const std::string & description )
```

Add metadata for one string register resource.
Resource group: beginResourceGroup() must have been called before calling this function. The added resource is automatically added to the last group added by beginResourceGroup().
Type: The added resource is of type 'string'. Use addRegister() to add a register of type 'numeric'.
The returned builder object is only valid until another resource is added. It is only intended to modify the resource that was added last.

Parameters

name	Name of the register. This is the same as the 'name' field of ResourceInfo.
description	Human readable description of the resource. This is the same as the 'description' field of ResourceInfo.

Returns

A RegisterBuilder object that can be used to set additional metadata for this register resource.

7.2.2.6 beginResourceGroup()

```
void iris::IrisInstanceBuilder::beginResourceGroup (
            const std::string & name,
            const std::string & description,
            uint64_t subRscIdStart = IRIS_UINT64_MAX,
            const std::string & cname = std::string() )
```

Begin a new resource group.
This has the following effects:

- Add a resource group if it does not yet exist. (If it already exists under 'name' all other parameters are ignored.)
- Assign all resources that are added by subsequent addRegister() or addParameter() calls to this group.

This function must be called before the first resource is added.

Parameters

name	Name of the resource group.
description	Description of the resource group.
subRscIdStart	If not IRIS_UINT64_MAX, start counting from this subRscId when new resources are added.
cname	C identifier-style name to use for this group if it is different from *name*.

See also

addParameter

addStringParameter

addRegister

addStringRegister

addNoValueRegister

7.2.2.7 enhanceParameter()

```
ParameterBuilder iris::IrisInstanceBuilder::enhanceParameter (
            ResourceId rscId )  [inline]
```

Get ParameterBuilder to enhance a parameter.
This function can be used to add/set meta info to an existing parameter. There is no strong use case for this function as all meta info can be set/added by using chained calls to the set...()/add...() functions directly after adding the parameter.
Usage: irisInstance.getBuilder().enhanceParameter(rscId).setFoo(...).setBar(...);
The returned builder object is only valid until another resource is added. It is only intended to modify the specified resource and to add fields to this resource.

Parameters

rscId	ResourceId of the parameter which is to be modified.

Returns

A ParameterBuilder object that can be used to set additional metadata for this parameter.

7.2.2.8 enhanceRegister()

```
RegisterBuilder iris::IrisInstanceBuilder::enhanceRegister (
            ResourceId rscId )  [inline]
```

Get RegisterBuilder to enhance register.
This function can be used to add sub-fields to register fields which is not possible in a chained call. The rscId can be retreieved by using getRscId() in the chained call. This function does not add any resource and does not modify any state.
Usage: irisInstance.getBuilder().enhanceRegister(rscId).setFoo(...).setBar(...).addField(...);
See DummyComponent.h for an example.
The returned builder object is only valid until another resource is added. It is only intended to modify the specified resource and to add fields to this resource.

Parameters

rscId	ResourceId of the resource which is to be modified or to which fields are to be added.

Returns

A RegisterBuilder object that can be used to set additional metadata for this resource.

7.2.2.9 getResourceInfo()

```
const ResourceInfo & iris::IrisInstanceBuilder::getResourceInfo (
```

```
            ResourceId rscId )  [inline]
```

Get ResourceInfo of a previously added register.
The returned reference will only be valid until more resources are added.

Parameters

rsc↩ Id	Resource Id of the resource.

7.2.2.10 setDefaultResourceDelegates()

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) READER,
IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &) WRITER>
void iris::IrisInstanceBuilder::setDefaultResourceDelegates (
            T * instance )  [inline]
```

Set both read and write resource delegates if they are defined in the same class.

See also

setDefaultResourceReadDelegate

setDefaultResourceWriteDelegate

Template Parameters

T	Class that defines resource read and write delegate methods.
READER	A method of class T which is a resource read delegate.
WRITER	A method of class T which is a resource write delegate.

Parameters

instance	An instance of class T on which READER and WRITER should be called.

7.2.2.11 setDefaultResourceReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
void iris::IrisInstanceBuilder::setDefaultResourceReadDelegate ( )  [inline]
```

Set default read access function for all subsequently added resources.
Resources that do not explicitly override the access function using
`addRegister(...).setReadDelegate(...)`
will use this delegate.
Usage: Pass in a global function to delegate resource reading to that function:

```
iris::IrisErrorCode myReadFunction(const iris::ResourceInfo &resourceInfo,
                                   iris::ResourceReadResult &result);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultReadDelegate<myReadFunction>();
builder->addRegister(...); // Uses myReadFunction
```

Template Parameters

FUNC	A function which is a resource read delegate.

7.2.2.12 setDefaultResourceReadDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDefaultResourceReadDelegate (
            ResourceReadDelegate delegate = ResourceReadDelegate() )  [inline]
```

Set default read access function for all subsequently added resources.
Resources that do not explicitly override the access function using
`addRegister(...).setReadDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all resources.
Usage: Pass an instance of ResourceReadDelegate into this function to delegate reading to any class T:

```
class MyClass
{
    ...
    iris::IrisErrorCode myReadFunction(const iris::ResourceInfo &resourceInfo,
                                       iris::ResourceReadResult &result);
};
MyClass myInstanceOfMyClass;
ResourceReadDelegate delegate =
    ResourceReadDelegate::make<MyClass, &MyClass::myReadFunction>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultReadDelegate(delegate);
builder->addRegister(...); // Uses myReadFunction
```

Parameters

delegate	Delegate object which will be called to read resources.

7.2.2.13 setDefaultResourceReadDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
void iris::IrisInstanceBuilder::setDefaultResourceReadDelegate (
            T * instance )  [inline]
```

Set default read access function for all subsequently added resources.
Resources that do not explicitly override the access function using
`addRegister(...).setReadDelegate(...)`
will use this delegate.
Usage: Pass an instance of class T where T::METHOD() is a resource read method:

```
class MyClass
{
    ...
    iris::IrisErrorCode myReadFunction(const iris::ResourceInfo &resourceInfo,
                                       iris::ResourceReadResult &result);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultReadDelegate<MyClass, &MyClass::myReadFunction>(myInstanceOfMyClass);
builder->addRegister(...); // Uses myReadFunction
```

Template Parameters

T	Class that defines a resource read delegate method.
METHOD	A method of class T which is a resource read delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.2.2.14 setDefaultResourceWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
```

```
void iris::IrisInstanceBuilder::setDefaultResourceWriteDelegate ( )  [inline]
```

Set default write access function for all subsequently added resources.
Resources that do not explicitly override the access function using
`addRegister(...).setWriteDelegate(...)`
will use this delegate.
Usage: Pass in a global function to delegate resource writing to that function:

```
iris::IrisErrorCode myWriteFunction(const iris::ResourceInfo &resourceInfo, const uint64_t *data);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultWriteDelegate<myWriteFunction>();
builder->addRegister(...); // Uses myWriteFunction
```

Template Parameters

FUNC	A function that is a resource write delegate.

7.2.2.15 setDefaultResourceWriteDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDefaultResourceWriteDelegate (
            ResourceWriteDelegate delegate = ResourceWriteDelegate() )  [inline]
```

Set default write access function for all subsequently added resources.
Resources that do not explicitly override the access function using
`addRegister(...).setWriteDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all resources.
Usage: Pass an instance of class T where T::METHOD() is a resource write method:

```
class MyClass
{
    ...
    iris::IrisErrorCode myWriteFunction(const iris::ResourceInfo &resourceInfo, const uint64_t *data);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
iris::ResourceWriteDelegate delegate =
    iris::ResourceWriteDelegate::make<MyClass, &MyClass::myWriteFunction>(myInstanceOfMyClass);
builder->setDefaultWriteDelegate(delegate);
builder->addRegister(...); // Uses myWriteFunction
```

Parameters

delegate	Delegate object which will be called to write resources.

7.2.2.16 setDefaultResourceWriteDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &)
METHOD>
void iris::IrisInstanceBuilder::setDefaultResourceWriteDelegate (
            T * instance )  [inline]
```

Set default write access function for all subsequently added resources.
Resources that do not explicitly override the access function using
`addRegister(...).setWriteDelegate(...)`
will use this delegate.
Usage: Pass an instance of class T where T::METHOD() is a resource write method:

```
class MyClass
{
    ...
    iris::IrisErrorCode myWriteFunction(const iris::ResourceInfo &resourceInfo, const uint64_t *data);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultWriteDelegate<MyClass, &MyClass::myWriteFunction>(myInstanceOfMyClass);
builder->addRegister(...); // Uses myWriteFunction
```

Template Parameters

T	Class that defines a resource write delegate method.
METHOD	A method of class T which is a resource write delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.2.2.17 setNextSubRscId()

```
void iris::IrisInstanceBuilder::setNextSubRscId (
            uint64_t nextSubRscId )  [inline]
```

Set the rscId that will be used for the next resource to be added.
Resources that are added following this call are assigned subRscIds starting at nextSubRscId.

Parameters

nextSubRscId	The subRscId that is used for the next resource to be added.

7.2.2.18 setPropertyCanonicalRnScheme()

```
void iris::IrisInstanceBuilder::setPropertyCanonicalRnScheme (
            const std::string & canonicalRnScheme )
```

Set the register.canonicalRnScheme instance property.
This property is visible in the list of properties returned by instance_getProperties().
This property defines the scheme used by the 'canonicalRn' member of the RegisterInfo object. This should be called upon initialization, before other instances have a chance to call instance_getProperties().
When using the function setCanonicalRnElfDwarf() the property is set automatically to "ElfDwarf" and it is not necessary to call this function.
When not calling setCanonicalRn() for any register it is not necessary to call this function. In this case the property will not exist which is ok.
Custom scheme names (other than ElfDwarf) should always be of the form <comnapy-name>.com/<scheme-name> to avoid conflicts.

Parameters

canonicalRnScheme	Name of the canonical register number scheme used by this instance.

7.2.2.19 setTag()

```
void iris::IrisInstanceBuilder::setTag (
            ResourceId rscId,
            const std::string & tag )
```

Set a tag for a specific resource.

Parameters

rscId	Resource Id for the resource that will have this tag set.

Parameters

tag	Name of the boolean tag that will be set to true.

See also

ResourceBuilder::setTag

RegisterBuilder::setTag

7.3 IrisInstanceBuilder event APIs

Set up event source metadata and event stream delegates.

Classes

- class iris::IrisInstanceBuilder::EventSourceBuilder

  Used to set metadata on an EventSource.

Functions

- EventSourceBuilder iris::IrisInstanceBuilder::addEventSource (const std::string &name, bool isHidden=false)

  Add metadata for an event source.
- EventSourceBuilder iris::IrisInstanceBuilder::addEventSource (const std::string &name, IrisEventEmitterBase &event_emitter, bool isHidden=false)

  Add metadata for an event source that uses an IrisEventEmitter.
- void iris::IrisInstanceBuilder::deleteEventSource (const std::string &name)

  Delete event source.
- EventSourceBuilder iris::IrisInstanceBuilder::enhanceEventSource (const std::string &name)

  Enhance existing event source.
- void iris::IrisInstanceBuilder::finalizeRegisterReadEvent ()
- void iris::IrisInstanceBuilder::finalizeRegisterUpdateEvent ()

  Finalize set up of an IrisEventEmitter.
- IrisInstanceEvent * iris::IrisInstanceBuilder::getIrisInstanceEvent ()
- bool iris::IrisInstanceBuilder::hasEventSource (const std::string &name)

  Check whether event source already exists.
- void iris::IrisInstanceBuilder::renameEventSource (const std::string &name, const std::string &newName)

  Rename existing event source.
- void iris::IrisInstanceBuilder::resetRegisterReadEvent ()

  Reset the active register read event.
- void iris::IrisInstanceBuilder::resetRegisterUpdateEvent ()

  Reset the active register update event.
- template<IrisErrorCode(*)(EventStream *&, const EventSourceInfo &, const std::vector< std::string > &) FUNC> void iris::IrisInstanceBuilder::setDefaultEsCreateDelegate ()

  Set the delegate that helps to create a new event stream for the simulation-specific event.
- void iris::IrisInstanceBuilder::setDefaultEsCreateDelegate (EventStreamCreateDelegate delegate)

  Set the delegate that helps to create a new event stream for the simulation-specific event.
- template<typename T , IrisErrorCode(T::*)(EventStream *&, const EventSourceInfo &, const std::vector< std::string > &) METHOD> void iris::IrisInstanceBuilder::setDefaultEsCreateDelegate (T *instance)

  Set the delegate that helps to create a new event stream for the simulation-specific event.
- EventSourceBuilder iris::IrisInstanceBuilder::setRegisterReadEvent (const std::string &name, const std::string &description=std::string())

  Add a new register read event source.

- EventSourceBuilder iris::IrisInstanceBuilder::setRegisterReadEvent (const std::string &name, IrisRegister↩ EventEmitterBase &event_emitter)

  Add a new register read event source.
- EventSourceBuilder iris::IrisInstanceBuilder::setRegisterUpdateEvent (const std::string &name, const std↩ ::string &description=std::string())

  Add a new register update event source.
- EventSourceBuilder iris::IrisInstanceBuilder::setRegisterUpdateEvent (const std::string &name, Iris↩ RegisterEventEmitterBase &event_emitter)

  Add a new register update event source.

7.3.1 Detailed Description

Set up event source metadata and event stream delegates.

7.3.2 Function Documentation

7.3.2.1 addEventSource() [1/2]

```
EventSourceBuilder iris::IrisInstanceBuilder::addEventSource (
            const std::string & name,
            bool isHidden = false )  [inline]
```

Add metadata for an event source.
Consider using addEventSource(const std::string& name, IrisEventEmitterBase& event_emitter) instead. Only use this if you want to implement a non-trivial trace source with its own event emitter handling.

Parameters

name	The name of the new event source.
isHidden	If true, the event source is hidden.

See also

EventSourceBuilder::setHidden

Returns

An EventSourceBuilder object that can be used to set additional attributes for this event source. The returned EventSourceBuilder is only valid until the next call to addEventSource().

7.3.2.2 addEventSource() [2/2]

```
EventSourceBuilder iris::IrisInstanceBuilder::addEventSource (
            const std::string & name,
            IrisEventEmitterBase & event_emitter,
            bool isHidden = false )  [inline]
```

Add metadata for an event source that uses an IrisEventEmitter.

Parameters

name	The name of the new event source.
event_emitter	The IrisEventEmitter for this event source.
isHidden	If true, the event source is hidden.

See also

EventSourceBuilder::setHidden

Returns

An EventSourceBuilder object that can be used to set additional attributes for this event source. The returned EventSourceBuilder is only valid until the next call to addEventSource(), setRegisterReadEvent(), or setRegisterWriteEvent().

7.3.2.3 deleteEventSource()

```
void iris::IrisInstanceBuilder::deleteEventSource (
            const std::string & name )  [inline]
```

Delete event source.

Parameters

name	The name of the event source.

7.3.2.4 enhanceEventSource()

```
EventSourceBuilder iris::IrisInstanceBuilder::enhanceEventSource (
            const std::string & name )  [inline]
```

Enhance existing event source.

Parameters

name	The name of the event source.

Returns

An EventSourceBuilder object that can be used to set additional attributes for this event source. The returned EventSourceBuilder is only valid until the next call to addEventSource(), setRegisterReadEvent(), or setRegisterWriteEvent().

7.3.2.5 finalizeRegisterReadEvent()

```
void iris::IrisInstanceBuilder::finalizeRegisterReadEvent ( )
```

Finalize the setup of an IrisEventEmitter.
When all the registers associated with all the read events have been added, call finalizeRegisterReadEvent() to add the event sources to the IrisInstance.

7.3.2.6 finalizeRegisterUpdateEvent()

```
void iris::IrisInstanceBuilder::finalizeRegisterUpdateEvent ( )
```

Finalize set up of an IrisEventEmitter.
When all the registers associated with all the write events have been added, call finalizeRegisterUpdateEvent() to add the event sources to the IrisInstance.

7.3.2.7 getIrisInstanceEvent()

```
IrisInstanceEvent * iris::IrisInstanceBuilder::getIrisInstanceEvent ( )  [inline]
```

Direct access to IrisInstanceEvent.
Do not use! This will be removed! Use the event api of IrisInstanceBuilder instead. This is a temporary hack.

7.3.2.8 hasEventSource()

```
bool iris::IrisInstanceBuilder::hasEventSource (
            const std::string & name )  [inline]
```

Check whether event source already exists.

Parameters

name	The name of the event source.

Returns

True iff the event source already exists.

7.3.2.9 renameEventSource()

```
void iris::IrisInstanceBuilder::renameEventSource (
            const std::string & name,
            const std::string & newName )  [inline]
```

Rename existing event source.

Parameters

name	The old name of the event source.
newName	The new name of the event source.

7.3.2.10 resetRegisterReadEvent()

```
void iris::IrisInstanceBuilder::resetRegisterReadEvent ( )
```

Reset the active register read event.
setRegisterReadEvent and resetRegisterReadEvent should be called in pair to scope the registers being added to be associated with a certain read event.

7.3.2.11 resetRegisterUpdateEvent()

```
void iris::IrisInstanceBuilder::resetRegisterUpdateEvent ( )
```

Reset the active register update event.
setRegisterUpdateEvent and resetRegisterUpdateEvent should be called in pair to scope the registers being added to be associated with a certain update event.

7.3.2.12 setDefaultEsCreateDelegate() [1/3]

```
template<IrisErrorCode(*)(EventStream *&, const EventSourceInfo &, const std::vector< std↩
::string > &) FUNC>
void iris::IrisInstanceBuilder::setDefaultEsCreateDelegate ( )  [inline]
```

Set the delegate that helps to create a new event stream for the simulation-specific event.
Consider using addEventSource(const std::string& name, IrisEventEmitterBase& event_emitter) instead. Only use this if you want to implement a non-trivial trace source with its own event emitter handling.
Event sources that do not explicitly override the access function using
`addEventSource(...).setEventStreamCreateDelegate(...)`
use this delegate.
Usage: Pass in a global function to which to delegate event stream creation:

```
iris::IrisErrorCode createEventStream(iris::EventStream*&, const iris::EventSourceInfo&,
                                      const std::vector<std::string>&)>
builder->setDefaultEsCreateDelegate<&MyClass::createEventStream>();
builder->addEventSource(...); // Uses createEventStream
```

Template Parameters

FUNC	Global function to which to delegate event stream creation.

7.3.2.13 setDefaultEsCreateDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDefaultEsCreateDelegate (
            EventStreamCreateDelegate delegate )  [inline]
```

Set the delegate that helps to create a new event stream for the simulation-specific event.
Consider using addEventSource(const std::string& name, IrisEventEmitterBase& event_emitter) instead. Only use this if you want to implement a non-trivial trace source with its own event emitter handling.
Event sources that do not explicitly override the access function using
`addEventSource(...).setEventStreamCreateDelegate(...)`
use this delegate.
Usage: Pass an instance of class T where T::METHOD() is an event stream creation method:

```
class MyClass
{
    ...
    iris::IrisErrorCode createEventStream(iris::EventStream*&, const iris::EventSourceInfo&,
                                          const std::vector<std::string>&)>
};
MyClass myInstanceOfMyClass;
EventStreamCreateDelegate delegate = EventStreamCreateDelegate::make<MyClass,
       &MyClass::createEventStream>(myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultEsCreateDelegateC(delegate);
builder->addEventSource(...); // Uses createEventStream
```

Parameters

delegate	Delegate object that will be called to create an event stream.

7.3.2.14 setDefaultEsCreateDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(EventStream *&, const EventSourceInfo &, const std↩
::vector< std::string > &) METHOD>
void iris::IrisInstanceBuilder::setDefaultEsCreateDelegate (
            T * instance )  [inline]
```

Set the delegate that helps to create a new event stream for the simulation-specific event.
Consider using addEventSource(const std::string& name, IrisEventEmitterBase& event_emitter) instead. Only use this if you want to implement a non-trivial trace source with its own event emitter handling.
Event sources that do not explicitly override the access function using
`addEventSource(...).setEventStreamCreateDelegate(...)`
use this delegate.
Usage: Pass an instance of class T where T::METHOD() is an event stream creation method:

```
class MyClass
{
    ...
    iris::IrisErrorCode createEventStream(iris::EventStream*&, const iris::EventSourceInfo&,
                                          const std::vector<std::string>&)>
};
MyClass myInstanceOfMyClass;
builder->setDefaultEsCreateDelegate<MyClass, &MyClass::createEventStream>(myInstanceOfMyClass);
builder->addEventSource(...); // Uses createEventStream
```

Template Parameters

T	Class that defines an event stream creation method.
METHOD	A method of class T which is an event stream creation method.

Parameters

instance	The instance of class T on which METHOD should be called.

7.3.2.15 setRegisterReadEvent() [1/2]

```
EventSourceBuilder iris::IrisInstanceBuilder::setRegisterReadEvent (
            const std::string & name,
            const std::string & description = std::string() )
```

Add a new register read event source.
Any registers added after calling setRegisterReadEvent() and before the next call to setRegisterReadEvent() or finalizeRegisterReadEvent() are associated with this event.
A call to setRegisterReadEvent() implicitly calls finalizeRegisterReadEvent() on the event source with name `name` iff an event emitter object (type IrisRegisterEventEmitterBase) is provided as an argument.
If the register read event source already exists (identified by name), the active register read event source simply switches to it.
Register read events have three standard fields:

Field	Description
REGISTER	The Iris rscId of the register accessed.
DEBUG	True if the read originated from a debug access.
VALUE	The value that was read.

Parameters

name	Name of the event source.
description	Description of the event source.

Returns

An EventSourceBuilder for the event allowing extra custom fields to be added.

7.3.2.16 setRegisterReadEvent() [2/2]

```
EventSourceBuilder iris::IrisInstanceBuilder::setRegisterReadEvent (
            const std::string & name,
            IrisRegisterEventEmitterBase & event_emitter )
```

Add a new register read event source.
Any registers added after calling setRegisterReadEvent() and before the next call to setRegisterReadEvent() or finalizeRegisterReadEvent() are associated with this event.
A call to setRegisterReadEvent() implicitly calls finalizeRegisterReadEvent() on the event source with name `name` iff an event emitter object (type IrisRegisterEventEmitterBase) is provided as an argument.
If the register read event source already exists (identified by name), the active register read event source simply switches to it.
Register read events have three standard fields:

Field	Description
REGISTER	The Iris rscId of the register accessed.
DEBUG	True if the read originated from a debug access.
VALUE	The value that was read.

Parameters

name	Name of the event source.
event_emitter	The event_emitter to associate with this event source.

Returns

An EventSourceBuilder for the event allowing extra custom fields to be added.

7.3.2.17 setRegisterUpdateEvent() [1/2]

```
EventSourceBuilder iris::IrisInstanceBuilder::setRegisterUpdateEvent (
            const std::string & name,
            const std::string & description = std::string() )
```

Add a new register update event source.
Any registers added after calling setRegisterUpdateEvent() and before the next call to setRegisterUpdateEvent() or finalizeRegisterUpdateEvent() are associated with this event.
A call to setRegisterUpdateEvent implicitly calls finalizeRegisterUpdateEvent() on the event source with name `name` iff an event emitter object (type IrisRegisterEventEmitterBase) is provided as an argument.
If the register update event source (identified by name) already exists, the active register update event source simply switches to it.
Register update events have four standard fields:

Field	Description
REGISTER	The Iris rscId of the register accessed.
DEBUG	True if the update originated from a debug access.
OLD_VALUE	The value that would have been read before the access was made.
NEW_VALUE	The value that would be read after the access was made.

Parameters

name	Name of the event source.
description	Description of the event source.

Returns

An EventSourceBuilder for the event allowing extra custom fields to be added.

7.3.2.18 setRegisterUpdateEvent() [2/2]

```
EventSourceBuilder iris::IrisInstanceBuilder::setRegisterUpdateEvent (
            const std::string & name,
            IrisRegisterEventEmitterBase & event_emitter )
```

Add a new register update event source.
Any registers added after calling setRegisterUpdateEvent() and before the next call to setRegisterUpdateEvent() or finalizeRegisterUpdateEvent() are associated with this event.
A call to setRegisterUpdateEvent implicitly calls finalizeRegisterUpdateEvent() on the event source with name `name` iff an event emitter object (type IrisRegisterEventEmitterBase) is provided as an argument.
If the register update event source (identified by name) already exists, the active register update event source simply switches to it.
Register update events have four standard fields:

Field	Description
REGISTER	The Iris rscId of the register accessed.
DEBUG	True if the update originated from a debug access.
OLD_VALUE	The value that would have been read before the access was made.
NEW_VALUE	The value that would be read after the access was made.

Parameters

name	Name of the event source.
event_emitter	The event_emitter to associate with this event source.

Returns

An EventSourceBuilder for the event allowing extra custom fields to be added.

7.4 IrisInstanceBuilder breakpoint APIs

Set up breakpoint hit notifications and breakpoint delegates.

Functions

- void **iris::IrisInstanceBuilder::addBreakpointCondition** (const std::string &name, const std::string &type, const std::string &description, const std::vector< std::string > bpt_types=std::vector< std::string >())

  Add an optional component-specific condition.
- const BreakpointInfo * iris::IrisInstanceBuilder::getBreakpointInfo (BreakpointId bptId)

  Get the breakpoint information for a given breakpoint.
- void iris::IrisInstanceBuilder::notifyBreakpointHit (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId)

  Notify clients that a code breakpoint was hit.
- void iris::IrisInstanceBuilder::notifyBreakpointHitData (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId, uint64_t accessAddr, uint64_t accessSize, const std::string &accessRw, const std::vector< uint64_t > &data)

  Notify clients that a data breakpoint was hit (`IRIS_BREAKPOINT_HIT`).
- void iris::IrisInstanceBuilder::notifyBreakpointHitRegister (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId, const std::string &accessRw, const std::vector< uint64_t > &data)

  Notify clients that a register breakpoint was hit (`IRIS_BREAKPOINT_HIT`).
- template<IrisErrorCode(*)(const BreakpointInfo &) FUNC>
  void iris::IrisInstanceBuilder::setBreakpointDeleteDelegate ()

  Set the delegate that is called when a breakpoint is deleted.
- void iris::IrisInstanceBuilder::setBreakpointDeleteDelegate (BreakpointDeleteDelegate delegate)

  Set the delegate that is called when a breakpoint is deleted.
- template<typename T , IrisErrorCode(T::*)(const BreakpointInfo &) METHOD>
  void iris::IrisInstanceBuilder::setBreakpointDeleteDelegate (T *instance)

  Set the delegate that is called when a breakpoint is deleted.
- template<IrisErrorCode(*)(BreakpointInfo &) FUNC>
  void iris::IrisInstanceBuilder::setBreakpointSetDelegate ()

  Set the delegate that is called when a breakpoint is set.
- void iris::IrisInstanceBuilder::setBreakpointSetDelegate (BreakpointSetDelegate delegate)

  Set the delegate that is called when a breakpoint is set.
- template<typename T , IrisErrorCode(T::*)(BreakpointInfo &) METHOD>
  void iris::IrisInstanceBuilder::setBreakpointSetDelegate (T *instance)

  Set the delegate that is called when a breakpoint is set.

- void iris::IrisInstanceBuilder::setHandleBreakpointHitsDelegate (std::function< IrisErrorCode(const BreakpointHitInfos &hitBpts)> delegate)

  Set the delegate that is called when a breakpoint is hit.

7.4.1 Detailed Description

Set up breakpoint hit notifications and breakpoint delegates.

7.4.2 Function Documentation

7.4.2.1 getBreakpointInfo()

```
const BreakpointInfo * iris::IrisInstanceBuilder::getBreakpointInfo (
            BreakpointId bptId )  [inline]
```

Get the breakpoint information for a given breakpoint.

Parameters

bptId	The breakpoint id of the breakpoint for which information is being requested.

Returns

The breakpoint information for the requested breakpoint. This returns nullptr if *bptId* is invalid.

7.4.2.2 notifyBreakpointHit()

```
void iris::IrisInstanceBuilder::notifyBreakpointHit (
            BreakpointId bptId,
            uint64_t time,
            uint64_t pc,
            MemorySpaceId pcSpaceId )  [inline]
```

Notify clients that a code breakpoint was hit.
This emits an (IRIS_BREAKPOINT_HIT) event.

Parameters

bptId	Breakpoint id for the breakpoint that was hit.
time	Simulation time at which the breakpoint was hit.
pc	Value of the program counter when the breakpoint was hit.
pcSpaceId	Memory space id for the PC when the breakpoint was hit.

7.4.2.3 notifyBreakpointHitData()

```
void iris::IrisInstanceBuilder::notifyBreakpointHitData (
            BreakpointId bptId,
            uint64_t time,
            uint64_t pc,
            MemorySpaceId pcSpaceId,
            uint64_t accessAddr,
            uint64_t accessSize,
```

```
            const std::string & accessRw,
            const std::vector< uint64_t > & data )  [inline]
```

Notify clients that a data breakpoint was hit (IRIS_BREAKPOINT_HIT).
This emits an (IRIS_BREAKPOINT_HIT) event.

Parameters

bptId	Breakpoint id for the breakpoint that was hit.
time	Simulation time at which the breakpoint was hit.
pc	Value of the program counter when the breakpoint was hit.
pcSpaceId	Memory space id for the PC when the breakpoint was hit.
accessAddr	Address of the access that hit.
accessSize	Size in bytes of the access that hit.
accessRw	Access direction. Should be "r" for a read access or "w" for a write access.
data	The data transferred by the access that hit.

7.4.2.4 notifyBreakpointHitRegister()

```
void iris::IrisInstanceBuilder::notifyBreakpointHitRegister (
            BreakpointId bptId,
            uint64_t time,
            uint64_t pc,
            MemorySpaceId pcSpaceId,
            const std::string & accessRw,
            const std::vector< uint64_t > & data )  [inline]
```

Notify clients that a register breakpoint was hit (IRIS_BREAKPOINT_HIT).
This emits an (IRIS_BREAKPOINT_HIT) event.

Parameters

bptId	Breakpoint id for the breakpoint that was hit.
time	Simulation time at which the breakpoint was hit.
pc	Value of the program counter when the breakpoint was hit.
pc↩ SpaceId	Memory space id for the PC when the breakpoint was hit.
accessRw	Access direction. Should be "r" for a read access or "w" for a write access.
data	The data transferred by the access that hit.

7.4.2.5 setBreakpointDeleteDelegate() [1/3]

```
template<IrisErrorCode(*)(const BreakpointInfo &) FUNC>
void iris::IrisInstanceBuilder::setBreakpointDeleteDelegate ( )  [inline]
```

Set the delegate that is called when a breakpoint is deleted.
Usage: Pass in a global function to call when a breakpoint is deleted:

```
iris::IrisErrorCode deleteBreakpoint(const iris::BreakpointInfo&);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setBreakpointDeleteDelegate<&deleteBreakpoint>();
```

Template Parameters

FUNC	Global function to call when a breakpoint is deleted.

7.4.2.6 setBreakpointDeleteDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setBreakpointDeleteDelegate (
            BreakpointDeleteDelegate delegate )  [inline]
```

Set the delegate that is called when a breakpoint is deleted.
Usage: Pass a breakpoint delete delegate:

```
class MyClass
{
    ...
    iris::IrisErrorCode deleteBreakpoint(const iris::BreakpointInfo&);
};
MyClass myInstanceOfMyClass;
BreakpointDeleteDelegate delegate = BreakpointDeleteDelegate::make<MyClass,
      &MyClass::deleteBreakpoint>(myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setBreakpointDeleteDelegate(delegate);
```

Parameters

delegate	Delegate object which will be called to delete a breakpoint.

7.4.2.7 setBreakpointDeleteDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const BreakpointInfo &) METHOD>
void iris::IrisInstanceBuilder::setBreakpointDeleteDelegate (
            T * instance )  [inline]
```

Set the delegate that is called when a breakpoint is deleted.
Usage: Pass an instance of class T, where T::METHOD() is a breakpoint delete delegate:

```
class MyClass
{
    ...
    iris::IrisErrorCode deleteBreakpoint(const iris::BreakpointInfo&);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setBreakpointDeleteDelegate<MyClass, &MyClass::deleteBreakpoint>(myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a breakpoint delete method.
METHOD	A method of class T which is a breakpoint delete delegate method.

Parameters

instance	The instance of class T on which METHOD should be called.

7.4.2.8 setBreakpointSetDelegate() [1/3]

```
template<IrisErrorCode(*)(BreakpointInfo &) FUNC>
void iris::IrisInstanceBuilder::setBreakpointSetDelegate ( )  [inline]
```

Set the delegate that is called when a breakpoint is set.
Usage: Pass in a global function to call when a breakpoint is set:

```
iris::IrisErrorCode setBreakpoint(iris::BreakpointInfo&);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setBreakpointSetDelegate<&setBreakpoint>();
```

Template Parameters

Template Parameters

FUNC	Global function to call when a breakpoint is set.

7.4.2.9 setBreakpointSetDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setBreakpointSetDelegate (
            BreakpointSetDelegate delegate )  [inline]
```

Set the delegate that is called when a breakpoint is set.
Usage: Pass a breakpoint set delegate:

```
class MyClass
{
    ...
    iris::IrisErrorCode setBreakpoint(iris::BreakpointInfo&);
};
MyClass myInstanceOfMyClass;
BreakpointSetDelegate delegate = BreakpointSetDelegate::make<MyClass,
       &MyClass::setBreakpoint>(myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setBreakpointSetDelegate(delegate);
```

Parameters

delegate	Delegate object which will be called to set a breakpoint.

7.4.2.10 setBreakpointSetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(BreakpointInfo &) METHOD>
void iris::IrisInstanceBuilder::setBreakpointSetDelegate (
            T * instance )  [inline]
```

Set the delegate that is called when a breakpoint is set.
Usage: Pass an instance of class T, where T::METHOD() is a breakpoint set delegate:

```
class MyClass
{
    ...
    iris::IrisErrorCode setBreakpoint(iris::BreakpointInfo&);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setBreakpointSetDelegate<MyClass, &MyClass::setBreakpoint>(myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a breakpoint set method.
METHOD	A method of class T which is a breakpoint set delegate method.

Parameters

instance	The instance of class T on which METHOD should be called.

7.4.2.11 setHandleBreakpointHitsDelegate()

```
void iris::IrisInstanceBuilder::setHandleBreakpointHitsDelegate (
            std::function< IrisErrorCode(const BreakpointHitInfos &hitBpts)> delegate )  [inline]
```

Set the delegate that is called when a breakpoint is hit.
Usage: Pass a handle breakpoint hit delegate:

```
class MyClass
{
    ...
    iris::IrisErrorCode handleBreakpointHits(const iris::BreakpointHitInfos&);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setHandleBreakpointHitsDelegate([&myInstanceOfMyClass] (const iris::BreakpointHitInfos& bptsHit)
                                         { return myInstanceOfMyClass.handleBreakpointHits(bptsHit) });
```

Parameters

delegate	Delegate object which will be called to handle a breakpoint hit.

7.5 IrisInstanceBuilder memory APIs

Set up address translation and memory space metadata and delegates.

Classes

- class iris::IrisInstanceBuilder::AddressTranslationBuilder

  Used to set metadata for an address translation.
- class iris::IrisInstanceBuilder::MemorySpaceBuilder

  Used to set metadata for a memory space.

Functions

- AddressTranslationBuilder iris::IrisInstanceBuilder::addAddressTranslation (MemorySpaceId inSpaceId, MemorySpaceId outSpaceId, const std::string &description)

  Add an address translation.
- MemorySpaceBuilder iris::IrisInstanceBuilder::addMemorySpace (const std::string &name)

  Add metadata for one memory space.
- template<IrisErrorCode(∗)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &) FUNC>
  void iris::IrisInstanceBuilder::setDefaultAddressTranslateDelegate ()

  Set the default address translation function for all subsequently added memory spaces.
- void iris::IrisInstanceBuilder::setDefaultAddressTranslateDelegate (MemoryAddressTranslateDelegate delegate=MemoryAddressTranslateDelegate())

  Set the default address translation function for all subsequently added memory spaces.
- template<typename T , IrisErrorCode(T::∗)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &) METHOD>
  void iris::IrisInstanceBuilder::setDefaultAddressTranslateDelegate (T ∗instance)

  Set the default address translation function for all subsequently added memory spaces.
- template<IrisErrorCode(∗)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std::vector< std::string > &, IrisValueMap &) FUNC>
  void iris::IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate ()

  Set the default sideband info function for all subsequently added memory spaces.
- void iris::IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate (MemoryGetSidebandInfoDelegate delegate)

  Set the default sideband info function for all subsequently added memory spaces.
- template<typename T , IrisErrorCode(T::∗)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std::vector< std::string > &, IrisValueMap &) METHOD>
  void iris::IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate (T ∗instance)

  Set the default sideband info function for all subsequently added memory spaces.
- template<IrisErrorCode(∗)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) FUNC>
  void iris::IrisInstanceBuilder::setDefaultMemoryReadDelegate ()

Set the default read function for all subsequently added memory spaces.

- void iris::IrisInstanceBuilder::setDefaultMemoryReadDelegate (MemoryReadDelegate delegate=MemoryReadDelegate())

  Set the default read function for all subsequently added memory spaces.

- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) METHOD>
  void iris::IrisInstanceBuilder::setDefaultMemoryReadDelegate (T *instance)

  Set the default read function for all subsequently added memory spaces.

- template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) FUNC>
  void iris::IrisInstanceBuilder::setDefaultMemoryWriteDelegate ()

  Set default write function for all subsequently added memory spaces.

- void iris::IrisInstanceBuilder::setDefaultMemoryWriteDelegate (MemoryWriteDelegate delegate=MemoryWriteDelegate())

  Set the default write function for all subsequently added memory spaces.

- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) METHOD>
  void iris::IrisInstanceBuilder::setDefaultMemoryWriteDelegate (T *instance)

  Set the default write function for all subsequently added memory spaces.

- void iris::IrisInstanceBuilder::setPropertyCanonicalMsnScheme (const std::string &canonicalMsnScheme)

  Set the memory.canonicalMsnScheme instance property.

7.5.1 Detailed Description

Set up address translation and memory space metadata and delegates.

7.5.2 Function Documentation

7.5.2.1 addAddressTranslation()

```
AddressTranslationBuilder iris::IrisInstanceBuilder::addAddressTranslation (
            MemorySpaceId inSpaceId,
            MemorySpaceId outSpaceId,
            const std::string & description )  [inline]
```

Add an address translation.
Add metadata for the address translation from the memory space indicated by *inSpaceId* to the memory space indicated by *outSpaceId*.
By explicitly adding an address translation using this function, the Iris instance can tell clients which address translations are supported and a component can provide a specific delegate function to perform that translation.

Parameters

inSpaceId	Memory space id for the input memory space of this translation.
out↩ SpaceId	Memory space id for the output memory space of this translation.
description	A human readable description of this translation. return An AddressTranslationBuilder object which allows additional configuration of this translation.

7.5.2.2 addMemorySpace()

```
MemorySpaceBuilder iris::IrisInstanceBuilder::addMemorySpace (
            const std::string & name )  [inline]
```

Add metadata for one memory space.
Typical use pattern:

```
addMemorySpace("name")
    .setDescription("description")
    .setMinAddr(...)
    .setMaxAddr(...)
    .setEndianness(...)
    .addAttribute(...)
    .addAttributeDefault(...);
```

Parameters

name	Name of the memory space to add.

Returns

A MemorySpaceBuilder object which can be used to configure metadata for the memory space.

7.5.2.3 setDefaultAddressTranslateDelegate() [1/3]

```
template<IrisErrorCode(*)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &)
FUNC>
void iris::IrisInstanceBuilder::setDefaultAddressTranslateDelegate ( )  [inline]
```

Set the default address translation function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
`addMemorySpace(...).setTranslationDelegate(...)`
will use this delegate.
Usage:

```
iris::IrisErrorCode translateAddress(MemorySpaceId inSpaceId, uint64_t address,
                                     MemorySpaceId outSpaceId,
                                     iris::MemoryAddressTranslationResult &result);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultAddressTranslateDelegate<&translateAddress>();
builder->addMemorySpace(...); // Uses translateAddress
```

Template Parameters

FUNC	Global function to call to translate addresses.

7.5.2.4 setDefaultAddressTranslateDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDefaultAddressTranslateDelegate (
            MemoryAddressTranslateDelegate delegate = MemoryAddressTranslateDelegate() )
[inline]
```

Set the default address translation function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
`addMemorySpace(...).setTranslationDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode translateAddress(MemorySpaceId inSpaceId, uint64_t address,
                                         MemorySpaceId outSpaceId,
                                         iris::MemoryAddressTranslationResult &result);
};
MyClass myInstanceOfMyClass;
iris::MemoryAddressTranslateDelegate delegate =
    iris::MemoryAddressTranslateDelegate::make<MyClass, &MyClass::translateAddress>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultAddressTranslateDelegate(delegate);
builder->addMemorySpace(...); // Uses translateAddress
```

Parameters

delegate	Delegate object which will be called to translate addresses.

7.5.2.5 setDefaultAddressTranslateDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslation↩
Result &) METHOD>
void iris::IrisInstanceBuilder::setDefaultAddressTranslateDelegate (
             T * instance )  [inline]
```

Set the default address translation function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
`addMemorySpace(...).setTranslationDelegate(...)`
will use this delegate.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode translateAddress(MemorySpaceId inSpaceId, uint64_t address,
                                         MemorySpaceId outSpaceId,
                                         iris::MemoryAddressTranslationResult &result);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultAddressTranslateDelegate<MyClass, &MyClass::translateAddress>(&myInstanceOfMyClass);
builder->addMemorySpace(...); // Uses translateAddress
```

Template Parameters

T	Class that defines an address translation delegate method.
METHOD	A method of class T which is an address translation delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.5.2.6 setDefaultGetMemorySidebandInfoDelegate() [1/3]

```
template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std↩
::vector< std::string > &, IrisValueMap &) FUNC>
void iris::IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate ( )  [inline]
```

Set the default sideband info function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the sideband function using
`addMemorySpace(...).setSidebandDelegate(...)`
will use this delegate.
Usage:

```
iris::IrisErrorCode getSidebandInfo(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                    const iris::IrisValueMap &attrib,
                                    const std::vector<std::string> &request,
                                    iris::IrisValueMap &result);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultGetMemorySidebandInfoDelegate<&getSidebandInfo>();
builder->addMemorySpace(...); // Uses getSidebandInfo
```

Template Parameters

FUNC	Global function to call to get sideband info.

7.5.2.7 setDefaultGetMemorySidebandInfoDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate (
            MemoryGetSidebandInfoDelegate delegate )  [inline]
```

Set the default sideband info function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the sideband function using
addMemorySpace(...).setSidebandDelegate(...)
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getSidebandInfo(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                        const iris::IrisValueMap &attrib,
                                        const std::vector<std::string> &request,
                                        iris::IrisValueMap &result);
};
MyClass myInstanceOfMyClass;
iris::MemoryAddressTranslateDelegate delegate =
    iris::MemoryAddressTranslateDelegate::make<MyClass, &MyClass::getSidebandInfo>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultGetMemorySidebandInfoDelegate(delegate);
builder->addMemorySpace(...); // Uses getSidebandInfo
```

Parameters

delegate	Delegate object which will be called to get sideband info.

7.5.2.8 setDefaultGetMemorySidebandInfoDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, const IrisValue↩
Map &, const std::vector< std::string > &, IrisValueMap &) METHOD>
void iris::IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate (
            T * instance )  [inline]
```

Set the default sideband info function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the sideband function using
addMemorySpace(...).setSidebandDelegate(...)
will use this delegate.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getSidebandInfo(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                        const iris::IrisValueMap &attrib,
                                        const std::vector<std::string> &request,
                                        iris::IrisValueMap &result);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultGetMemorySidebandInfoDelegate<MyClass, &MyClass::getSidebandInfo>(&myInstanceOfMyClass);
builder->addMemorySpace(...); // Uses getSidebandInfo
```

Template Parameters

T	Class that defines a sideband info delegate method.
METHOD	A method of class T which is a sideband info delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.5.2.9 setDefaultMemoryReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) FUNC>
void iris::IrisInstanceBuilder::setDefaultMemoryReadDelegate ( )  [inline]
```

Set the default read function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
`addMemorySpace(...).setReadDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_not_implemented for all requests.
Usage: Pass an instance of class T, where T::METHOD() is a memory read method:

```
iris::IrisErrorCode readMemory(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                               uint64_t byteWidth, uint64_t count,
                               const iris::IrisValueMap &attrib,
                               iris::MemoryReadResult &result);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultMemoryReadDelegate<readMemory>();
builder->addMemorySpace(...); // Uses readMemory
```

Template Parameters

FUNC	A memory read delegate function.

7.5.2.10 setDefaultMemoryReadDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDefaultMemoryReadDelegate (
              MemoryReadDelegate delegate = MemoryReadDelegate() )  [inline]
```

Set the default read function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
`addMemorySpace(...).setReadDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_not_implemented for all requests.
Usage: Pass an instance of class T, where T::METHOD() is a memory read method:

```
class MyClass
{
    ...
    iris::IrisErrorCode readMemory(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                   uint64_t byteWidth, uint64_t count,
                                   const iris::IrisValueMap &attrib,
                                   iris::MemoryReadResult &result);
};
MyClass myInstanceOfMyClass;
iris::MemoryReadDelegate delegate =
    iris::MemoryReadDelegate::make<MyClass, &MyClass::readMemory>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultMemoryReadDelegate(delegate);
builder->addMemorySpace(...); // Uses readMemory
```

Parameters

delegate	Delegate object which will be called to read memory.

7.5.2.11 setDefaultMemoryReadDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) METHOD>
void iris::IrisInstanceBuilder::setDefaultMemoryReadDelegate (
              T * instance )  [inline]
```

Set the default read function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using

addMemorySpace(...).setReadDelegate(...)
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩
not_implemented for all requests.
Usage: Pass an instance of class T, where T::METHOD() is a memory read method:

```
class MyClass
{
    ...
    iris::IrisErrorCode readMemory(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                   uint64_t byteWidth, uint64_t count,
                                   const iris::IrisValueMap &attrib,
                                   iris::MemoryReadResult &result);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultMemoryReadDelegate<MyClass, &MyClass::readMemory>(myInstanceOfMyClass);
builder->addMemorySpace(...); // Uses readMemory
```

Template Parameters

T	Class that defines a memory read delegate method.
METHOD	A method of class T which is a memory read delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.5.2.12 setDefaultMemoryWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const Attribute↩
ValueMap &, const uint64_t *, MemoryWriteResult &) FUNC>
void iris::IrisInstanceBuilder::setDefaultMemoryWriteDelegate ( )  [inline]
```

Set default write function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
addMemorySpace(...).setWriteDelegate(...)
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩
not_implemented for all requests.
Usage: Pass an instance of class T, where T::METHOD() is a memory read method:

```
iris::IrisErrorCode writeMemory(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                uint64_t byteWidth, uint64_t count,
                                const iris::IrisValueMap &attrib,
                                const uint64_t *data,
                                iris::MemoryWriteResult &result);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultMemoryWriteDelegate<&writeMemory>();
builder->addMemorySpace(...); // Uses writeMemory
```

Template Parameters

FUNC	Global function to call to write memory.

7.5.2.13 setDefaultMemoryWriteDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDefaultMemoryWriteDelegate (
            MemoryWriteDelegate delegate = MemoryWriteDelegate() )  [inline]
```

Set the default write function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
addMemorySpace(...).setWriteDelegate(...)
will use this delegate.

Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩
not_implemented for all requests.
Usage: Pass an instance of class T, where T::METHOD() is a memory read method:

```
class MyClass
{
    ...
    iris::IrisErrorCode writeMemory(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                    uint64_t byteWidth, uint64_t count,
                                    const iris::IrisValueMap &attrib,
                                    const uint64_t *data,
                                    iris::MemoryWriteResult &result);
};
MyClass myInstanceOfMyClass;
iris::MemoryReadDelegate delegate =
    iris::MemoryWriteDelegate::make<MyClass, &MyClass::writeMemory>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultMemoryWriteDelegate(delegate);
builder->addMemorySpace(...); // Uses writeMemory
```

Parameters

delegate	Delegate object which will be called to write memory.

7.5.2.14 setDefaultMemoryWriteDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64↩
_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) METHOD>
void iris::IrisInstanceBuilder::setDefaultMemoryWriteDelegate (
             T * instance )  [inline]
```

Set the default write function for all subsequently added memory spaces.
Memory spaces that do not explicitly override the access function using
`addMemorySpace(...).setWriteDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩
not_implemented for all requests.
Usage: Pass an instance of class T, where T::METHOD() is a memory read method:

```
class MyClass
{
    ...
    iris::IrisErrorCode writeMemory(const iris::MemorySpaceInfo &spaceInfo, uint64_t address,
                                    uint64_t byteWidth, uint64_t count,
                                    const iris::IrisValueMap &attrib,
                                    const uint64_t *data,
                                    iris::MemoryWriteResult &result);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultMemoryWriteDelegate<MyClass, &MyClass::writeMemory>(&myInstanceOfMyClass);
builder->addMemorySpace(...); // Uses writeMemory
```

Template Parameters

T	Class that defines a memory read delegate method.
METHOD	A method of class T which is a memory read delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.5.2.15 setPropertyCanonicalMsnScheme()

```
void iris::IrisInstanceBuilder::setPropertyCanonicalMsnScheme (
             const std::string & canonicalMsnScheme )
```

Set the memory.canonicalMsnScheme instance property.
This property is visible in the list of properties returned by instance_getProperties().
This property defines the scheme used by the 'canonicalMsn' member of the MemorySpaceInfo object. The default is 'arm.com/memoryspaces' which is used by all Arm components. This default can be overridden by calling this function. This should be called upon initialisation, before other instances have a chance to call instance_get↩Properties().

Parameters

canonicalMsnScheme	Name of the canonical memory space number scheme used by this instance.

7.6 IrisInstanceBuilder image loading APIs

Set up image-loading delegates.

Functions

- template<IrisErrorCode(∗)(const std::vector< uint8_t > &) FUNC>
  void iris::IrisInstanceBuilder::setLoadImageDataDelegate ()

  Set the delegate to load an image from the data provided.
- void iris::IrisInstanceBuilder::setLoadImageDataDelegate (ImageLoadDataDelegate delegate=ImageLoadDataDelegate())

  Set the delegate to load an image from the data provided.
- template<typename T , IrisErrorCode(T::∗)(const std::vector< uint8_t > &) METHOD>
  void iris::IrisInstanceBuilder::setLoadImageDataDelegate (T ∗instance)

  Set the delegate to load an image from the data provided.
- template<IrisErrorCode(∗)(const std::string &) FUNC>
  void iris::IrisInstanceBuilder::setLoadImageFileDelegate ()

  Set the delegate to load an image from a file.
- void iris::IrisInstanceBuilder::setLoadImageFileDelegate (ImageLoadFileDelegate delegate=ImageLoadFileDelegate())

  Set the delegate to load an image from a file.
- template<typename T , IrisErrorCode(T::∗)(const std::string &) METHOD>
  void iris::IrisInstanceBuilder::setLoadImageFileDelegate (T ∗instance)

  Set the delegate to load an image from a file.

7.6.1 Detailed Description

Set up image-loading delegates.

7.6.2 Function Documentation

7.6.2.1 setLoadImageDataDelegate() [1/3]

```
template<IrisErrorCode(*)(const std::vector< uint8_t > &) FUNC>
void iris::IrisInstanceBuilder::setLoadImageDataDelegate ( )  [inline]
```

Set the delegate to load an image from the data provided.
Usage:

```
    iris::IrisErrorCode loadImageData(const std::vector<uint64_t> &data, uint64_t dataSizeInBytes);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setLoadImageDataDelegate<&loadImageData>();
```

Template Parameters

FUNC	Global function to call for image loading.

7.6.2.2 setLoadImageDataDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setLoadImageDataDelegate (
            ImageLoadDataDelegate delegate = ImageLoadDataDelegate() )  [inline]
```

Set the delegate to load an image from the data provided.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode loadImageData(const std::vector<uint64_t> &data, uint64_t dataSizeInBytes);
};
MyClass myInstanceOfMyClass;
iris::MemoryAddressTranslateDelegate delegate =
    iris::MemoryAddressTranslateDelegate::make<MyClass, &MyClass::loadImageData>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setLoadImageDataDelegate(delegate);
```

Parameters

delegate	Delegate object to call for image loading.

7.6.2.3 setLoadImageDataDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const std::vector< uint8_t > &) METHOD>
void iris::IrisInstanceBuilder::setLoadImageDataDelegate (
            T * instance )  [inline]
```

Set the delegate to load an image from the data provided.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode loadImageData(const std::vector<uint64_t> &data, uint64_t dataSizeInBytes);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setLoadImageDataDelegate<MyClass, &MyClass::loadImageData>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines an image-loading delegate method.
METHOD	A method of class T which is an image-loading delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.6.2.4 setLoadImageFileDelegate() [1/3]

```
template<IrisErrorCode(*)(const std::string &) FUNC>
void iris::IrisInstanceBuilder::setLoadImageFileDelegate ( )  [inline]
```

Set the delegate to load an image from a file.
Usage:

```
iris::IrisErrorCode loadImageFile(const std::string &path);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setLoadImageFileDelegate<&loadImageFile>();
```

Template Parameters

FUNC	Global function to call for image loading.

7.6.2.5 setLoadImageFileDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setLoadImageFileDelegate (
            ImageLoadFileDelegate delegate = ImageLoadFileDelegate() )  [inline]
```

Set the delegate to load an image from a file.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode loadImageFile(const std::string &path);
};
MyClass myInstanceOfMyClass;
iris::MemoryAddressTranslateDelegate delegate =
    iris::MemoryAddressTranslateDelegate::make<MyClass, &MyClass::loadImageFile>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setLoadImageFileDelegate(delegate);
```

Parameters

delegate	Delegate object to call for image loading.

7.6.2.6 setLoadImageFileDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const std::string &) METHOD>
void iris::IrisInstanceBuilder::setLoadImageFileDelegate (
            T * instance )  [inline]
```

Set the delegate to load an image from a file.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode loadImageFile(const std::string &path);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setLoadImageFileDelegate<MyClass, &MyClass::loadImageFile>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines an image-loading delegate method.
METHOD	A method of class T which is an image-loading delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.7 IrisInstanceBuilder image readData callback APIs.

Open images for reading.

Functions

- uint64_t iris::IrisInstanceBuilder::openImage (const std::string &filename)

  Open an image to be read using image_loadDataPull() or image_loadDataRead().

7.7.1 Detailed Description

Open images for reading.

7.7.2 Function Documentation

7.7.2.1 openImage()

```
uint64_t iris::IrisInstanceBuilder::openImage (
            const std::string & filename )  [inline]
```

Open an image to be read using image_loadDataPull() or image_loadDataRead().

Parameters

filename	The name of the file to be read.

Returns

The tag number to use when calling image_loadDataPull().

7.8 IrisInstanceBuilder execution stepping APIs

Set up delegates to set and get the step count and the remaining steps.

Functions

- template<IrisErrorCode(*)(uint64_t &, const std::string &) FUNC>
  void iris::IrisInstanceBuilder::setRemainingStepGetDelegate ()

  Set the delegate to get the remaining steps for this instance.
- void iris::IrisInstanceBuilder::setRemainingStepGetDelegate (RemainingStepGetDelegate delegate)

  Set the delegate to get the remaining steps for this instance.
- template<typename T , IrisErrorCode(T::*)(uint64_t &, const std::string &) METHOD>
  void iris::IrisInstanceBuilder::setRemainingStepGetDelegate (T *instance)

  Set the delegate to get the remaining steps for this instance.
- template<IrisErrorCode(*)(uint64_t, const std::string &) FUNC>
  void iris::IrisInstanceBuilder::setRemainingStepSetDelegate ()

  Set the delegate to set the remaining steps for this instance.
- void iris::IrisInstanceBuilder::setRemainingStepSetDelegate (RemainingStepSetDelegate delegate=RemainingStepSetDelegat

  Set the delegate to set the remaining steps for this instance.
- template<typename T , IrisErrorCode(T::*)(uint64_t, const std::string &) METHOD>
  void iris::IrisInstanceBuilder::setRemainingStepSetDelegate (T *instance)

  Set the delegate to set the remaining steps for this instance.
- template<IrisErrorCode(*)(uint64_t &, const std::string &) FUNC>
  void iris::IrisInstanceBuilder::setStepCountGetDelegate ()

  Set the delegate to get the step count for this instance.
- void iris::IrisInstanceBuilder::setStepCountGetDelegate (StepCountGetDelegate delegate=StepCountGetDelegate())

  Set the delegate to get the step count for this instance.
- template<typename T , IrisErrorCode(T::*)(uint64_t &, const std::string &) METHOD>
  void iris::IrisInstanceBuilder::setStepCountGetDelegate (T *instance)

  Set the delegate to get the step count for this instance.

7.8.1 Detailed Description

Set up delegates to set and get the step count and the remaining steps.

7.8.2 Function Documentation

7.8.2.1 setRemainingStepGetDelegate() [1/3]

```
template<IrisErrorCode(*)(uint64_t &, const std::string &) FUNC>
void iris::IrisInstanceBuilder::setRemainingStepGetDelegate ( )  [inline]
```

Set the delegate to get the remaining steps for this instance.
Usage:

```
iris::IrisErrorCode getRemainingSteps(uint64_t &steps, const std::string &unit);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setRemainingStepGetDelegate<&getRemainingSteps>();
```

Template Parameters

FUNC	Global function to call to get the remaining steps.

7.8.2.2 setRemainingStepGetDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setRemainingStepGetDelegate (
            RemainingStepGetDelegate delegate )  [inline]
```

Set the delegate to get the remaining steps for this instance.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getRemainingSteps(uint64_t &steps, const std::string &unit);
};
MyClass myInstanceOfMyClass;
iris::RemainingStepGetDelegate delegate =
    iris::RemainingStepGetDelegate::make<MyClass, &MyClass::getRemainingSteps>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setRemainingStepGetDelegate(delegate);
```

Parameters

delegate	Delegate object to call to get the remaining steps.

7.8.2.3 setRemainingStepGetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(uint64_t &, const std::string &) METHOD>
void iris::IrisInstanceBuilder::setRemainingStepGetDelegate (
            T * instance )  [inline]
```

Set the delegate to get the remaining steps for this instance.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getRemainingSteps(uint64_t &steps, const std::string &unit);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setRemainingStepGetDelegate<MyClass, &MyClass::getRemainingSteps>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a get remaining steps delegate method.
METHOD	A method of class T that is a get remaining steps delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.8.2.4 setRemainingStepSetDelegate() [1/3]

```
template<IrisErrorCode(*)(uint64_t, const std::string &) FUNC>
void iris::IrisInstanceBuilder::setRemainingStepSetDelegate ( )  [inline]
```

Set the delegate to set the remaining steps for this instance.
Usage:

```
iris::IrisErrorCode setRemainingSteps(uint64_t steps, const std::string &unit);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setRemainingStepSetDelegate<&setRemainingSteps>();
```

Template Parameters

FUNC	Global function to call to set the remaining steps.

7.8.2.5 setRemainingStepSetDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setRemainingStepSetDelegate (
            RemainingStepSetDelegate delegate = RemainingStepSetDelegate() )  [inline]
```

Set the delegate to set the remaining steps for this instance.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode setRemainingSteps(uint64_t steps, const std::string &unit);
};
MyClass myInstanceOfMyClass;
iris::RemainingStepSetDelegate delegate =
    iris::RemainingStepSetDelegate::make<MyClass, &MyClass::setRemainingSteps>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setRemainingStepSetDelegate(delegate);
```

Parameters

delegate	Delegate object to call to set the remaining steps.

7.8.2.6 setRemainingStepSetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(uint64_t, const std::string &) METHOD>
void iris::IrisInstanceBuilder::setRemainingStepSetDelegate (
            T * instance )  [inline]
```

Set the delegate to set the remaining steps for this instance.
Usage:

```
class MyClass
{
    ...
```

```
    iris::IrisErrorCode setRemainingSteps(uint64_t steps, const std::string &unit);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setRemainingStepSetDelegate<MyClass, &MyClass::setRemainingSteps>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a set remaining steps delegate method.
METHOD	A method of class T that is a set remaining steps delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.8.2.7 setStepCountGetDelegate() [1/3]

```
template<IrisErrorCode(*)(uint64_t &, const std::string &) FUNC>
void iris::IrisInstanceBuilder::setStepCountGetDelegate ( )  [inline]
```

Set the delegate to get the step count for this instance.
Usage:

```
iris::IrisErrorCode getStepCount(uint64_t &count, const std::string &unit);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setStepCountGetDelegate<&getStepCount>();
```

Template Parameters

FUNC	Global function to call to get the step count.

7.8.2.8 setStepCountGetDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setStepCountGetDelegate (
            StepCountGetDelegate delegate = StepCountGetDelegate() )  [inline]
```

Set the delegate to get the step count for this instance.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getStepCount(uint64_t &count, const std::string &unit);
};
MyClass myInstanceOfMyClass;
iris::StepCountGetDelegate delegate =
    iris::StepCountGetDelegate::make<MyClass, &MyClass::getStepCount>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setStepCountGetDelegate(delegate);
```

Parameters

delegate	Delegate object to call to get the step count.

7.8.2.9 setStepCountGetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(uint64_t &, const std::string &) METHOD>
void iris::IrisInstanceBuilder::setStepCountGetDelegate (
```

```
            T * instance )  [inline]
```

Set the delegate to get the step count for this instance.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getStepCount(uint64_t &count, const std::string &unit);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setStepCountGetDelegate<MyClass, &MyClass::getStepCount>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a get step count delegate method.
METHOD	A method of class T which is a get step count delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

7.9 Disassembler delegate functions

Set disassembler delegates.

Classes

- class iris::IrisInstanceDisassembler

  Disassembler add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< const std::vector< uint64_t > &, uint64_t, const std::string &, DisassembleContext &, DisassemblyLine & > iris::DisassembleOpcodeDelegate

  Get the disassembly for an individual opcode.
- typedef IrisDelegate< std::string & > iris::GetCurrentDisassemblyModeDelegate

  Get the current disassembly mode.

Functions

- void iris::IrisInstanceDisassembler::addDisassemblyMode (const std::string &name, const std::string &description)

  Add a disassembly mode.
- void iris::IrisInstanceDisassembler::attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- iris::IrisInstanceDisassembler::IrisInstanceDisassembler (IrisInstance *irisInstance=nullptr)

  Construct an IrisInstanceDisassembler.
- void iris::IrisInstanceDisassembler::setDisassembleOpcodeDelegate (DisassembleOpcodeDelegate delegate)

  Set the delegate to get the disassembly of Opcode.
- void iris::IrisInstanceDisassembler::setGetCurrentModeDelegate (GetCurrentDisassemblyModeDelegate delegate)

  Set the delegate to get the current disassembly mode.
- void iris::IrisInstanceDisassembler::setGetDisassemblyDelegate (std::function< IrisErrorCode(GetDisassemblyArgs &)> delegate)

  Set the delegate to get the disassembly of a chunk of memory.

Variables

- uint64_t **iris::GetDisassemblyArgs::address**
- AttributeValueMap **iris::GetDisassemblyArgs::attrib**
- uint64_t **iris::GetDisassemblyArgs::count**
- std::vector< DisassemblyLine > & **iris::GetDisassemblyArgs::disassemblyLineOut**
- uint64_t **iris::GetDisassemblyArgs::maxAddr**
- std::string **iris::GetDisassemblyArgs::mode**
- MemorySpaceId **iris::GetDisassemblyArgs::spaceId**

7.9.1 Detailed Description

Set disassembler delegates.

7.9.2 Typedef Documentation

7.9.2.1 DisassembleOpcodeDelegate

```
typedef IrisDelegate<const std::vector<uint64_t>&, uint64_t, const std::string&, DisassembleContext&, DisassemblyLine&> iris::DisassembleOpcodeDelegate
```

Get the disassembly for an individual opcode.

```
IrisErrorCode disassembleOpcode(const std::vector<uint64_t> &opcode, uint64_t address, const std::string
      &mode,
                                DisassembleContext &context, DisassemblyLine &disassemblyLineOut)
```

Error: Return E_* error code if it failed to disassemble.

7.9.2.2 GetCurrentDisassemblyModeDelegate

```
typedef IrisDelegate<std::string&> iris::GetCurrentDisassemblyModeDelegate
```

Get the current disassembly mode.

```
IrisErrorCode getCurrentMode(std::string &currentMode)
```

Error: Return E_* error code if it failed to get the current mode.

7.9.3 Function Documentation

7.9.3.1 addDisassemblyMode()

```
void iris::IrisInstanceDisassembler::addDisassemblyMode (
            const std::string & name,
            const std::string & description )
```

Add a disassembly mode.

This function should only be called during the initial setup of the instance, after which the list of disassembly modes should be static.

Parameters

name	Name of the mode being added.
description	Description of the mode being added.

7.9.3.2 attachTo()

```
void iris::IrisInstanceDisassembler::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

7.9.3.3 IrisInstanceDisassembler()

```
iris::IrisInstanceDisassembler::IrisInstanceDisassembler (
            IrisInstance * irisInstance = nullptr )
```

Construct an IrisInstanceDisassembler.

Parameters

irisInstance	IrisInstance to attach this add-on to.

7.9.3.4 setDisassembleOpcodeDelegate()

```
void iris::IrisInstanceDisassembler::setDisassembleOpcodeDelegate (
            DisassembleOpcodeDelegate delegate )  [inline]
```

Set the delegate to get the disassembly of Opcode.

Parameters

delegate	Delegate object that will be called to get the disassembly of an opcode.

7.9.3.5 setGetCurrentModeDelegate()

```
void iris::IrisInstanceDisassembler::setGetCurrentModeDelegate (
            GetCurrentDisassemblyModeDelegate delegate )  [inline]
```

Set the delegate to get the current disassembly mode.

Parameters

delegate	Delegate object that will be called to get the current disassembly mode.

7.9.3.6 setGetDisassemblyDelegate()

```
void iris::IrisInstanceDisassembler::setGetDisassemblyDelegate (
            std::function< IrisErrorCode(GetDisassemblyArgs &)> delegate )  [inline]
```

Set the delegate to get the disassembly of a chunk of memory.

Parameters

delegate	Delegate object that will be called to get the disassembly of a chunk of memory.

7.10 Semihosting data request flag constants

Flags used to define the behavior of the readData() method.

7.10.1 Detailed Description

Flags used to define the behavior of the readData() method.

Chapter 8

Class Documentation

8.1 iris::IrisInstanceBuilder::AddressTranslationBuilder Class Reference

Used to set metadata for an address translation.
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- **AddressTranslationBuilder** (IrisInstanceMemory::AddressTranslationInfoAndAccess &info_)
- template<IrisErrorCode(∗)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &) FUNC>
  AddressTranslationBuilder & setTranslateDelegate ()

  Set the delegate to perform an address translation.
- AddressTranslationBuilder & setTranslateDelegate (MemoryAddressTranslateDelegate delegate)

  Set the delegate to perform an address translation.
- template<typename T , IrisErrorCode(T::∗)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &) METHOD>
  AddressTranslationBuilder & setTranslateDelegate (T ∗instance)

  Set the delegate to perform an address translation.

8.1.1 Detailed Description

Used to set metadata for an address translation.

8.1.2 Member Function Documentation

8.1.2.1 setTranslateDelegate() [1/3]

```
template<IrisErrorCode(*)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &)
FUNC>
AddressTranslationBuilder & iris::IrisInstanceBuilder::AddressTranslationBuilder::setTranslate↩
Delegate ( )  [inline]
```

Set the delegate to perform an address translation.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultAddressTranslationDelegate

Template Parameters

FUNC	An address translation delegate function.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.1.2.2 setTranslateDelegate() [2/3]

```
AddressTranslationBuilder & iris::IrisInstanceBuilder::AddressTranslationBuilder::setTranslate↩
Delegate (
            MemoryAddressTranslateDelegate delegate )  [inline]
```

Set the delegate to perform an address translation.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultAddressTranslationDelegate

Parameters

delegate	MemoryAddressTranslateDelegate object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.1.2.3 setTranslateDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslation↩
Result &) METHOD>
AddressTranslationBuilder & iris::IrisInstanceBuilder::AddressTranslationBuilder::setTranslate↩
Delegate (
            T * instance )  [inline]
```

Set the delegate to perform an address translation.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultAddressTranslationDelegate

Template Parameters

T	A class that defines a method with the right signature to be a memory address translation delegate.
METHOD	A memory address translation delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.2 iris::IrisInstanceMemory::AddressTranslationInfoAndAccess Struct Reference

Contains static address translation information.
`#include <IrisInstanceMemory.h>`

Public Member Functions

- **AddressTranslationInfoAndAccess** (MemorySpaceId inSpaceId, MemorySpaceId outSpaceId, const std::string &description)

Public Attributes

- MemoryAddressTranslateDelegate **translateDelegate**
- MemorySupportedAddressTranslationResult **translationInfo**

8.2.1 Detailed Description

Contains static address translation information.
The documentation for this struct was generated from the following file:

- IrisInstanceMemory.h

8.3 iris::BreakpointHitInfo Struct Reference

Public Attributes

- const std::vector< uint64_t > **accessData**
- const BreakpointInfo & **bptInfo**
- const bool **isReadAccess**

The documentation for this struct was generated from the following file:

- IrisInstanceBreakpoint.h

8.4 iris::BreakpointHitInfos Struct Reference

Public Attributes

- std::vector< BreakpointHitInfo > **breakpointHitInfos** {}

The documentation for this struct was generated from the following file:

- IrisInstanceBreakpoint.h

8.5 iris::IrisInstanceBuilder::EventSourceBuilder Class Reference

Used to set metadata on an EventSource.
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- EventSourceBuilder & addEnumElement (const std::string &fieldName, uint64_t value, const std::string &symbol, const std::string &description="")

  Add an enum element to a specific field.
- EventSourceBuilder & addEnumElement (uint64_t value, const std::string &symbol, const std::string &description="")

Add an enum element for the last field added.

- EventSourceBuilder & addField (const std::string &name, const std::string &type, uint64_t sizeInBytes, const std::string &description)

  Add a field to this event source.

- template<typename T >
  EventSourceBuilder & addOption (const std::string &name, const std::string &type, const T &defaultValue, bool optional, const std::string &description)

  Declare an option for event streams of an event source.

- **EventSourceBuilder** (IrisInstanceEvent::EventSourceInfoAndDelegate &info_)
- EventSourceBuilder & hasSideEffects (bool hasSideEffects_=true)

  Set hasSideEffects for this event source.

- EventSourceBuilder & removeEnumElement (const std::string &fieldName, uint64_t value)

  Remove an enum element by value from a specific field.

- EventSourceBuilder & renameEnumElement (const std::string &fieldName, uint64_t value, const std::string &newEnumSymbol)

  Rename an enum element by value of a specific field.

- EventSourceBuilder & setCounter (bool counter=true)

  Set the `counter` field.

- EventSourceBuilder & setDescription (const std::string &description)

  Set the `description` field.

- EventSourceBuilder & setEventStreamCreateDelegate (EventStreamCreateDelegate delegate)

  Set the delegate to create an event stream.

- template<typename T , IrisErrorCode(T::*)(EventStream *&, const EventSourceInfo &, const std::vector< std::string > &) METHOD>
  EventSourceBuilder & setEventStreamCreateDelegate (T *instance)

  Set the delegate to create an event stream.

- EventSourceBuilder & setFormat (const std::string &format)

  Set the `format` field.

- EventSourceBuilder & setHidden (bool hidden=true)

  Hide/unhide this event source.

- EventSourceBuilder & setName (const std::string &name)

  Set the `name` field.

8.5.1 Detailed Description

Used to set metadata on an EventSource.

8.5.2 Member Function Documentation

8.5.2.1 addEnumElement() [1/2]

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::addEnumElement (
            const std::string & fieldName,
            uint64_t value,
            const std::string & symbol,
            const std::string & description = "" )  [inline]
```

Add an enum element to a specific field.

Parameters

fieldName	Field name.
value	The value of the enum element.
symbol	The symbol string that will be displayed instead of the value.
description	A human readable description of this enum.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.2 addEnumElement() [2/2]

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::addEnumElement (
            uint64_t value,
            const std::string & symbol,
            const std::string & description = "" )  [inline]
```

Add an enum element for the last field added.
This must be called after addField().

Parameters

value	The value of the enum element.
symbol	The symbol string that will be displayed instead of the value.
description	A human readable description of this enum.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.3 addField()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::addField (
            const std::string & name,
            const std::string & type,
            uint64_t sizeInBytes,
            const std::string & description )  [inline]
```

Add a field to this event source.
This method constructs an EventSourceFieldInfo object and adds it to the EventSource. It should be called multiple times to add multiple fields.

Parameters

name	The name of the field.
type	The type of the field.
sizeInBytes	The size of the field in bytes.
description	A human readable description of the field.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.4 addOption()

```
template<typename T >
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::addOption (
            const std::string & name,
            const std::string & type,
            const T & defaultValue,
```

```
            bool optional,
            const std::string & description )  [inline]
```

Declare an option for event streams of an event source.
This method fills the 'options' member of EventSourceInfo. It may be called multiple times to add multiple options.

Parameters

name	The name of the option.
type	The type of the option.
defaultValue	The default value of the option.
optional	True if the option is optional, False otherwise.
description	A human readable description of the option.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.5 hasSideEffects()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::hasSideEffects (
            bool hasSideEffects_ = true )  [inline]
```

Set hasSideEffects for this event source.

Parameters

*hasSideEffects_*	If true, this event source has side effects. This is exotic. Normal event sources do not have side effects. For example semihosting events have side effects.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.6 removeEnumElement()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::removeEnumElement (
            const std::string & fieldName,
            uint64_t value )  [inline]
```

Remove an enum element by value from a specific field.

Parameters

fieldName	Field name.
value	The value of the enum element.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.7 renameEnumElement()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::renameEnumElement (
            const std::string & fieldName,
```

```
            uint64_t value,
            const std::string & newEnumSymbol )  [inline]
```

Rename an enum element by value of a specific field.

Parameters

fieldName	Field name.
value	The value of the enum element.
newEnumSymbol	New enum symbol.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.8 setCounter()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::setCounter (
            bool counter = true )  [inline]
```

Set the `counter` field.

Parameters

counter	The counter field of the EventSourceInfo object.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.9 setDescription()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::setDescription (
            const std::string & description )  [inline]
```

Set the `description` field.

Parameters

description	The description field of the EventSourceInfo object.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.10 setEventStreamCreateDelegate() [1/2]

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::setEventStreamCreate↩
Delegate (
            EventStreamCreateDelegate delegate )  [inline]
```

Set the delegate to create an event stream.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultEsCreateDelegate

Parameters

delegate	EventStreamCreateDelegate object.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.11 setEventStreamCreateDelegate() [2/2]

```
template<typename T , IrisErrorCode(T::*)(EventStream *&, const EventSourceInfo &, const std↩
::vector< std::string > &) METHOD>
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::setEventStreamCreate↩
Delegate (
            T * instance )  [inline]
```

Set the delegate to create an event stream.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultEsCreateDelegate

Template Parameters

T	A class that defines a method with the right signature to be an event stream creation method.
METHOD	An event stream creation delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.12 setFormat()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::setFormat (
            const std::string & format )  [inline]
```

Set the `format` field.

Parameters

format	The format field of the EventSourceInfo object.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.13 setHidden()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::setHidden (
```

```
            bool hidden = true )  [inline]
```

Hide/unhide this event source.

Parameters

hidden	If true, this event source is not listed in event_getEventSources() calls but can still be accessed by event_getEventSource() for clients that know the event source's name.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

8.5.2.14 setName()

```
EventSourceBuilder & iris::IrisInstanceBuilder::EventSourceBuilder::setName (
            const std::string & name )  [inline]
```

Set the `name` field.

Parameters

name	The name field of the EventSourceInfo object.

Returns

A reference to this EventSourceBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.6 iris::IrisInstanceEvent::EventSourceInfoAndDelegate Struct Reference

Contains the metadata and delegates for a single EventSource.
`#include <IrisInstanceEvent.h>`

Public Attributes

- EventStreamCreateDelegate **createEventStream**
- EventSourceInfo **info**
- bool **isProxy** {false}
- bool **isValid** {true}
- ProxyEventInfo **proxyEventInfo**

8.6.1 Detailed Description

Contains the metadata and delegates for a single EventSource.
The documentation for this struct was generated from the following file:

- IrisInstanceEvent.h

8.7 iris::EventStream Class Reference

Base class for event streams.
`#include <IrisInstanceEvent.h>`
Inherited by iris::IrisEventStream.

Public Member Functions

- virtual IrisErrorCode action (const BreakpointAction &action_)

  Execute action on trace stream.
- void addField (const IrisU64StringConstant &field, bool value)

  Add a boolean field value.
- template<class T >
  void addField (const IrisU64StringConstant &field, const T &value)

  Add a field value.
- void addField (const IrisU64StringConstant &field, const uint8_t *data, size_t sizeInBytes)

  Add byte array.
- void addField (const IrisU64StringConstant &field, int64_t value)

  Add a sint field value.
- void addField (const IrisU64StringConstant &field, uint64_t value)

  Add a uint field value.
- void addFieldSlow (const std::string &field, bool value)

  Add a boolean field value.
- template<class T >
  void addFieldSlow (const std::string &field, const T &value)

  Add a field value.
- void addFieldSlow (const std::string &field, const uint8_t *data, size_t sizeInBytes)

  Add byte array.
- void addFieldSlow (const std::string &field, int64_t value)

  Add a sint field value.
- void addFieldSlow (const std::string &field, uint64_t value)

  Add a uint field value.
- bool checkRangePc (uint64_t pc) const

  Check the range for the PC.
- virtual IrisErrorCode disable ()=0

  Disable this event stream.
- void emitEventBegin (IrisRequest &req, uint64_t time, uint64_t pc=IRIS_UINT64_MAX)

  Start to emit an event callback.
- void emitEventBegin (uint64_t time, uint64_t pc=IRIS_UINT64_MAX)

  Start to emit an event callback.
- void emitEventEnd (bool send=true)

  Emit the callback.
- virtual IrisErrorCode enable ()=0

  Enable this event stream.
- **EventStream** ()

  Construct a new event stream.
- virtual IrisErrorCode flush (RequestId requestId)

  Flush event stream.
- uint64_t getCountVal () const

  Get the current value of the counter.
- InstanceId getEcInstId () const

  Get the event callback instance id for this event stream.
- EventStreamId getEsId () const

  Get the Id of this event stream.
- EventSourceId getEventSourceId () const

  Get the event source id of the event source of this event stream (not the event stream id)
- const EventSourceInfo * getEventSourceInfo () const

Get the event source info of this event stream.

- InstanceId getProxiedByInstanceId () const

  Get the instance ID of the Iris instance which is a proxy for this event stream.
- virtual IrisErrorCode getState (IrisValueMap &fields)

  Query the current state of the event.
- virtual IrisErrorCode **insertTrigger** ()
- bool isCounter () const

  Is this event stream a counter?
- bool isEnabled () const

  Is this event stream currently enabled?
- bool IsProxiedByOtherInstance () const

  Is there another Iris instance which is a proxy for this event stream?
- bool IsProxyForOtherInstance () const

  Is this event stream a proxy for an event stream in another Iris instance?
- void selfRelease ()

  Trigger the event stream to be released.
- void setCounter (uint64_t startVal, const EventCounterMode &counterMode)

  Set the counter mode and starting value for this event stream.
- virtual IrisErrorCode setOptions (const AttributeValueMap &options, bool eventStreamCreate, std::string &errorMessageOut)

  Set options.
- void setProperties (IrisInstance *irisInstance, IrisInstanceEvent *irisInstanceEvent, EventSourceId evSrcId, InstanceId ecInstId, const std::string &ecFunc, EventStreamId esId, bool syncEc)

  Initialize this event stream.
- void setProxiedByInstanceId (InstanceId instId)

  Saves the instance ID of the Iris instance that is a proxy for this event stream.
- void **setProxyForOtherInstance** ()

  Set that this event stream is a proxy for an event stream in another Iris instance.
- IrisErrorCode setRanges (const std::string &aspect, const std::vector< uint64_t > &ranges)

  Set the trace ranges for this event stream.

Protected Attributes

- std::string **aspect**

  — members for range —
- bool **aspectFound** {}

  Found aspect in one of the fields.
- bool counter {}

  — members for a counter —
- EventCounterMode **counterMode** {}

  Specified counter mode.
- uint64_t **curAspectValue** {}

  The current aspect value.
- uint64_t **curVal** {}
- std::string **ecFunc**

  The event callback function name specified by eventEnable().
- InstanceId **ecInstId** {IRIS_UINT64_MAX}

  Specify target instance that this event is sent to.
- bool **enabled** {}

  Event is only generated when the event stream is enabled.
- EventStreamId **esId** {IRIS_UINT64_MAX}

The event stream id.

- EventSourceId **evSrcId** {IRIS_UINT64_MAX}

  The event source of this stream.
- IrisU64JsonWriter::Object **fieldObj**
- IrisRequest * **internal_req** {}
- IrisInstance * irisInstance {}

  — basic members —
- IrisInstanceEvent * **irisInstanceEvent** {}

  Parent IrisInstanceEvent owning this stream.
- bool **isProxyForOtherInstance** {false}

  Is this event stream a proxy for an event stream in another Iris instance?
- InstanceId proxiedByInstanceId {IRIS_UINT64_MAX}
- std::vector< uint64_t > **ranges**
- IrisRequest * **req** {}

  Generate callback requests.
- uint64_t **startVal** {}

  Start value and current value for a counter.
- bool **syncEc** {}

  Synchronous callback behavior.

8.7.1 Detailed Description

Base class for event streams.
This class is abstract as it is not known how to enable or disable an event for a simulation.

8.7.2 Member Function Documentation

8.7.2.1 action()

```
virtual IrisErrorCode iris::EventStream::action (
            const BreakpointAction & action_ )  [inline], [virtual]
```

Execute action on trace stream.
This function is usually only ever called by breakpoints which have an action other than eventStream_enable or eventStream_disable.
This function is only implemented by very specific event streams.

Returns

An error code indicating whether the operation was successful.

8.7.2.2 addField() [1/5]

```
void iris::EventStream::addField (
            const IrisU64StringConstant & field,
            bool value )  [inline]
```

Add a boolean field value.
Fast variant for argument names up to 23 chars. Use this if you can. This will also record the aspect value if the aspect of range check is set.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.3 addField() [2/5]

```
template<class T >
void iris::EventStream::addField (
            const IrisU64StringConstant & field,
            const T & value )  [inline]
```

Add a field value.
This is supported for all types supported by IrisU64JsonWriter and IrisObjects.h. Fast variant for argument names up to 23 chars. Use this if you can.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.4 addField() [3/5]

```
void iris::EventStream::addField (
            const IrisU64StringConstant & field,
            const uint8_t * data,
            size_t sizeInBytes )  [inline]
```

Add byte array.
Fast variant for argument names up to 23 chars. Use this if you can.

Parameters

field	The name of the field whose value is set.
data	Pointer to byte data.
sizeInBytes	Size of byte data.

8.7.2.5 addField() [4/5]

```
void iris::EventStream::addField (
            const IrisU64StringConstant & field,
            int64_t value )  [inline]
```

Add a sint field value.
Fast variant for argument names up to 23 chars. Use this if you can. This will also record the aspect value if the aspect of range check is set.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.6 addField() [5/5]

```
void iris::EventStream::addField (
            const IrisU64StringConstant & field,
            uint64_t value )  [inline]
```

Add a uint field value.

Fast variant for argument names up to 23 chars. Use this if you can. This will also record the aspect value if the aspect of range check is set.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.7 addFieldSlow() [1/5]

```
void iris::EventStream::addFieldSlow (
            const std::string & field,
            bool value )  [inline]
```

Add a boolean field value.
Slow variant for argument names with more than 23 chars. Do not use unless you have to. This will also record the aspect value if the aspect of range check is set.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.8 addFieldSlow() [2/5]

```
template<class T >
void iris::EventStream::addFieldSlow (
            const std::string & field,
            const T & value )  [inline]
```

Add a field value.
This is supported for all types supported by IrisU64JsonWriter and IrisObjects.h. Slow variant for argument names with more than 23 chars. Do not use unless you have to.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.9 addFieldSlow() [3/5]

```
void iris::EventStream::addFieldSlow (
            const std::string & field,
            const uint8_t * data,
            size_t sizeInBytes )  [inline]
```

Add byte array.
Slow variant for argument names with more than 23 chars. Do not use unless you have to.

Parameters

field	The name of the field whose value is set.
data	Pointer to byte data.
sizeInBytes	Size of byte data.

8.7.2.10 addFieldSlow() [4/5]

```
void iris::EventStream::addFieldSlow (
            const std::string & field,
            int64_t value )  [inline]
```

Add a sint field value.
Slow variant for argument names with more than 23 chars. Do not use unless you have to. This will also record the aspect value if the aspect of range check is set.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.11 addFieldSlow() [5/5]

```
void iris::EventStream::addFieldSlow (
            const std::string & field,
            uint64_t value )  [inline]
```

Add a uint field value.
Slow variant for argument names with more than 23 chars. Do not use unless you have to. This will also record the aspect value if the aspect of range check is set.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.7.2.12 checkRangePc()

```
bool iris::EventStream::checkRangePc (
            uint64_t pc ) const  [inline]
```

Check the range for the PC.
This can optionally be called before generating the callback request (before calling emitEventBegin()).

Parameters

pc	The program counter value to check.

Returns

`true` if the PC value is in range or no range is configured, `false` otherwise.

8.7.2.13 disable()

```
virtual IrisErrorCode iris::EventStream::disable ( )  [pure virtual]
```

Disable this event stream.
This function is only called when isEnabled()/enabled == true. It is not necessary to verify this inside the disable() method.

Returns

An error code indicating whether the event stream was successfully disabled. This should be E_ok if it was disabled or E_error_disabling_event_stream if it could not be disabled.

Implemented in iris::IrisEventStream.

8.7.2.14 emitEventBegin() [1/2]

```
void iris::EventStream::emitEventBegin (
            IrisRequest & req,
            uint64_t time,
            uint64_t pc = IRIS_UINT64_MAX )
```

Start to emit an event callback.

Parameters

req	A request object to use to construct the event callback.
time	The time in simulation ticks at which the event occurred.
pc	The program counter value when the event occurred.

8.7.2.15 emitEventBegin() [2/2]

```
void iris::EventStream::emitEventBegin (
            uint64_t time,
            uint64_t pc = IRIS_UINT64_MAX )
```

Start to emit an event callback.

Parameters

time	The time in simulation ticks at which the event occurred.
pc	The program counter value when the event occurred.

8.7.2.16 emitEventEnd()

```
void iris::EventStream::emitEventEnd (
            bool send = true )
```

Emit the callback.
This will also check the ranges and maintain the counter.

Parameters

send	If `true`, event callbacks are sent to the callee immediately. If `false`, the callback are not sent immediately, allowing the caller to delay sending.

8.7.2.17 enable()

```
virtual IrisErrorCode iris::EventStream::enable ( )  [pure virtual]
```

Enable this event stream.
This function is only called when isEnabled()/enabled == false. It is not necessary to verify this inside the enable() method.

Returns

An error code indicating whether the event stream was successfully enabled. This should be E_ok if it was enabled or E_error_enabling_event_stream if it could not be enabled.

Implemented in iris::IrisEventStream.

8.7.2.18 flush()

```
virtual IrisErrorCode iris::EventStream::flush (
            RequestId requestId )  [inline], [virtual]
```

Flush event stream.
Supported in the derived classes for specific event sources.

Parameters

requestId	Request id of the eventStream_flush() call. This is returned to the caller in an extra FLUSH_REQUEST_ID field in the response to the flush call.

Returns

An error code indicating whether the operation was successful.

8.7.2.19 getCountVal()

```
uint64_t iris::EventStream::getCountVal ( ) const  [inline]
```

Get the current value of the counter.

Returns

The current value of the event counter.

8.7.2.20 getEcInstId()

```
InstanceId iris::EventStream::getEcInstId ( ) const  [inline]
```

Get the event callback instance id for this event stream.

Returns

The instId for the instance that this event stream calls when an event fires.

8.7.2.21 getEsId()

```
EventStreamId iris::EventStream::getEsId ( ) const  [inline]
```

Get the Id of this event stream.

Returns

The esId for this event stream.

8.7.2.22 getEventSourceId()

```
EventSourceId iris::EventStream::getEventSourceId ( ) const  [inline]
```

Get the event source id of the event source of this event stream (not the event stream id)

Returns

The event source id of this event stream.

8.7.2.23 getEventSourceInfo()

```
const EventSourceInfo * iris::EventStream::getEventSourceInfo ( ) const  [inline]
```
Get the event source info of this event stream.

Returns

The event source info that was used to create this event stream.

8.7.2.24 getProxiedByInstanceId()

```
InstanceId iris::EventStream::getProxiedByInstanceId ( ) const  [inline]
```
Get the instance ID of the Iris instance which is a proxy for this event stream.

Returns

`The` instance ID of the Iris instance which is a proxy

8.7.2.25 getState()

```
virtual IrisErrorCode iris::EventStream::getState (
            IrisValueMap & fields )  [inline], [virtual]
```
Query the current state of the event.
Supported in the derived classes for specific event sources.

Parameters

fields	A map which will be populated with the current values for this event's fields.

Returns

An error code indicating whether the operation was successful.

8.7.2.26 isCounter()

```
bool iris::EventStream::isCounter ( ) const  [inline]
```
Is this event stream a counter?

Returns

`true` if this event stream is a counter, otherwise `false`.

8.7.2.27 isEnabled()

```
bool iris::EventStream::isEnabled ( ) const  [inline]
```
Is this event stream currently enabled?

Returns

`true` if this event stream is enabled or `false` if it disabled.

8.7.2.28 IsProxiedByOtherInstance()

```
bool iris::EventStream::IsProxiedByOtherInstance ( ) const  [inline]
```
Is there another Iris instance which is a proxy for this event stream?

Returns

`true` if this event stream is being proxied by another Iris instance, otherwise `false`.

8.7.2.29 IsProxyForOtherInstance()

```
bool iris::EventStream::IsProxyForOtherInstance ( ) const  [inline]
```

Is this event stream a proxy for an event stream in another Iris instance?

Returns

`true` if this event stream is a proxy, otherwise `false`.

8.7.2.30 selfRelease()

```
void iris::EventStream::selfRelease ( )  [inline]
```

Trigger the event stream to be released.
If this event stream is not waiting for any response, release it immediately. Otherwise, release it when it has finished waiting. The event stream is disabled beforehand if it is still enabled.

Note

Do not touch anything related to this object after calling this function.

Do not call this function if this object was not created by 'new'.

8.7.2.31 setCounter()

```
void iris::EventStream::setCounter (
            uint64_t startVal,
            const EventCounterMode & counterMode )
```

Set the counter mode and starting value for this event stream.

Parameters

startVal	The starting value of the counter.
counterMode	The mode in which this counter operates.

8.7.2.32 setOptions()

```
virtual IrisErrorCode iris::EventStream::setOptions (
            const AttributeValueMap & options,
            bool eventStreamCreate,
            std::string & errorMessageOut )  [inline], [virtual]
```

Set options.
Supported in the derived classes for specific event sources. This is called by setProperties() which in turn is called when the event stream is created. Creating the event stream will fail when this function returns an error and when an options argument is present in eventStream_create().

Parameters

options	Map of options (key/value pairs).

Parameters

eventStreamCreate	True: These are the options set by eventStream_create(). False: These are options set by eventStream_setOptions().
errorMessageOut	When this function returns an error it should set errorMessageOut to a meaningful error message.

Returns

An error code indicating whether the operation was successful.

8.7.2.33 setProperties()

```
void iris::EventStream::setProperties (
            IrisInstance * irisInstance,
            IrisInstanceEvent * irisInstanceEvent,
            EventSourceId evSrcId,
            InstanceId ecInstId,
            const std::string & ecFunc,
            EventStreamId esId,
            bool syncEc )
```

Initialize this event stream.

Parameters

irisInstance	The IrisInstance that is producing this stream. This will be used to send event callback requests.
irisInstanceEvent	Parent IrisInstancEvent owning this event stream.
evSrcId	The metadata for the event source generating this stream.
ecInstId	The event callback instId: the instance that this stream calls when an event fires.
ecFunc	The event callback function: the function that is called when an event fires.
esId	The event stream id for this event stream.
syncEc	True if this event stream is synchronous and should send event callbacks as requests. If false event callbacks are sent as notifications and do not wait for a response.

8.7.2.34 setProxiedByInstanceId()

```
void iris::EventStream::setProxiedByInstanceId (
            InstanceId instId ) [inline]
```

Saves the instance ID of the Iris instance that is a proxy for this event stream.

Parameters

instId	The instance ID of the proxy Iris instance

8.7.2.35 setRanges()

```
IrisErrorCode iris::EventStream::setRanges (
            const std::string & aspect,
            const std::vector< uint64_t > & ranges )
```

Set the trace ranges for this event stream.

Parameters

aspect	The field whose range to check.
ranges	A list where each 3 elements form a 3-tuple of (mask, start, end) values to configure ranges.

Returns

An error code indicating whether the ranges could be set successfully.

8.7.3 Member Data Documentation

8.7.3.1 counter

```
bool iris::EventStream::counter {}  [protected]
```

— members for a counter —
Is a counter?

8.7.3.2 irisInstance

```
IrisInstance* iris::EventStream::irisInstance {}  [protected]
```

— basic members —
The Iris instance that created this event.

8.7.3.3 proxiedByInstanceId

```
InstanceId iris::EventStream::proxiedByInstanceId {IRIS_UINT64_MAX}  [protected]
```

An event stream in another Iris instance is a proxy for this event stream proxiedByInstanceId - the instance ID of the other Iris instance
The documentation for this class was generated from the following file:

- IrisInstanceEvent.h

8.8 iris::IrisInstanceBuilder::FieldBuilder Class Reference

Used to set metadata on a register field resource.
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- FieldBuilder & addEnum (const std::string &symbol, const IrisValue &value, const std::string &description=std::string())

  Add a symbol to the enums field for numeric resources.
- FieldBuilder addField (const std::string &name, uint64_t lsbOffset, uint64_t bitWidth, const std::string &description)

  Add another subregister field to the parent register.
- FieldBuilder addLogicalField (const std::string &name, uint64_t bitWidth, const std::string &description)

  Add another logical subregister field to the parent register.
- FieldBuilder & addStringEnum (const std::string &stringValue, const std::string &description=std::string())

  Add a symbol to the enums field for string resources.
- **FieldBuilder** (IrisInstanceResource::ResourceInfoAndAccess &info_, RegisterBuilder *parent_reg_, IrisInstanceBuilder *instance_builder_)
- ResourceId getRscId () const

Return the rscId that was allocated for this resource.

- FieldBuilder & getRscId (ResourceId &rscIdOut)

  Get the rscId that was allocated for this resource.
- RegisterBuilder & parent ()

  Get the RegisterBuilder for the parent register.
- FieldBuilder & setAddressOffset (uint64_t addressOffset)

  Set the `addressOffset` field.
- FieldBuilder & setBitWidth (uint64_t bitWidth)

  Set the `bitWidth` field.
- FieldBuilder & setBreakpointSupportInfo (const std::string &supported)

  Set the `breakpointSupport` field.
- FieldBuilder & setCanonicalRn (uint64_t canonicalRn_)

  Set the `canonicalRn` field.
- FieldBuilder & setCanonicalRnElfDwarf (uint16_t architecture, uint16_t dwarfRegNum)

  Set the `canonicalRn` field for "ElfDwarf" scheme.
- FieldBuilder & setCname (const std::string &cname)

  Set the `cname` field.
- FieldBuilder & **setDescr** (const std::string &description)

  Deprecated alias for setDescription().
- FieldBuilder & setDescription (const std::string &description)

  Set the `description` field.
- FieldBuilder & setFormat (const std::string &format)

  Set the `format` field.
- FieldBuilder & setLsbOffset (uint64_t lsbOffset)

  Set the `lsbOffset` field.
- FieldBuilder & setName (const std::string &name)

  Set the `name` field.
- FieldBuilder & setParentRscId (ResourceId parentRscId)

  Set the `parentRscId` field.
- template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
  FieldBuilder & setReadDelegate ()

  Set the delegate to read the resource.
- FieldBuilder & setReadDelegate (ResourceReadDelegate readDelegate)

  Set the delegate to read the resource.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
  FieldBuilder & setReadDelegate (T *instance)

  Set the delegate to read the resource.
- template<typename T >
  FieldBuilder & setResetData (std::initializer_list< T > &&t)

  Set the `resetData` field for wide registers.
- FieldBuilder & setResetData (uint64_t value)

  Set the `resetData` field to a value <= 64 bit.
- template<typename Container >
  FieldBuilder & setResetDataFromContainer (const Container &container)

  Set the `resetData` field for wide registers.
- FieldBuilder & setResetString (const std::string &resetString)

  Set the `resetString` field.
- FieldBuilder & setRwMode (const std::string &rwMode)

  Set the `rwMode` field.
- FieldBuilder & setSubRscId (uint64_t subRscId)

  Set the `subRscId` field.

- FieldBuilder & setTag (const std::string &tag)

  Set the named boolean tag to true (e.g. isPc)
- FieldBuilder & setTag (const std::string &tag, const IrisValue &value)

  Set a tag to the specified value.
- FieldBuilder & setType (const std::string &type)

  Set the `type` field.
- template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
  FieldBuilder & setWriteDelegate ()

  Set the delegate to write the resource.
- FieldBuilder & setWriteDelegate (ResourceWriteDelegate writeDelegate)

  Set the delegate to write the resource.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &) METHOD>
  FieldBuilder & setWriteDelegate (T *instance)

  Set the delegate to write the resource.
- template<typename T >
  FieldBuilder & setWriteMask (std::initializer_list< T > &&t)

  Set the `writeMask` field for wide registers.
- FieldBuilder & setWriteMask (uint64_t value)

  Set the `writeMask` field to a value <= 64 bit.
- template<typename Container >
  FieldBuilder & setWriteMaskFromContainer (const Container &container)

  Set the `writeMask` field for wide registers.

Protected Attributes

- IrisInstanceResource::ResourceInfoAndAccess * **info** {}
- IrisInstanceBuilder * **instance_builder** {}
- RegisterBuilder * **parent_reg** {}

8.8.1 Detailed Description

Used to set metadata on a register field resource.

8.8.2 Member Function Documentation

8.8.2.1 addEnum()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::addEnum (
            const std::string & symbol,
            const IrisValue & value,
            const std::string & description = std::string() )  [inline]
```

Add a symbol to the enums field for numeric resources.
This should be called multiple times to add multiple symbols.

Parameters

symbol	The symbol string to be associated with the specified value.
value	The value of this symbol.
description	A description of this symbol.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.2 addField()

```
FieldBuilder iris::IrisInstanceBuilder::FieldBuilder::addField (
            const std::string & name,
            uint64_t lsbOffset,
            uint64_t bitWidth,
            const std::string & description )  [inline]
```

Add another subregister field to the parent register.

See also

RegisterBuilder::addField

8.8.2.3 addLogicalField()

```
FieldBuilder iris::IrisInstanceBuilder::FieldBuilder::addLogicalField (
            const std::string & name,
            uint64_t bitWidth,
            const std::string & description )  [inline]
```

Add another logical subregister field to the parent register.

See also

RegisterBuilder::addField

8.8.2.4 addStringEnum()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::addStringEnum (
            const std::string & stringValue,
            const std::string & description = std::string() )  [inline]
```

Add a symbol to the enums field for string resources.
This should be called multiple times to add multiple symbols.

Parameters

value	The string value of this symbol. This is also used as the symbols string.
description	A description of this symbol.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.5 getRscId() [1/2]

```
ResourceId iris::IrisInstanceBuilder::FieldBuilder::getRscId ( ) const  [inline]
```

Return the rscId that was allocated for this resource.

Returns

The rscId that was allocated for this resource.

8.8.2.6 getRscId() [2/2]

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::getRscId (
            ResourceId & rscIdOut ) [inline]
```

Get the rscId that was allocated for this resource.
This variant is useful to get the ResourceId of fields added in a chained call where return values are not practical.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.7 parent()

```
RegisterBuilder & iris::IrisInstanceBuilder::FieldBuilder::parent ( ) [inline]
```

Get the RegisterBuilder for the parent register.

Returns

The RegisterBuilder object for the parent register.

8.8.2.8 setAddressOffset()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setAddressOffset (
            uint64_t addressOffset ) [inline]
```

Set the `addressOffset` field.

Parameters

addressOffset	The addressOffset field of the RegisterInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.9 setBitWidth()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setBitWidth (
            uint64_t bitWidth ) [inline]
```

Set the `bitWidth` field.

Parameters

bitWidth	The bitWidth field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.10 setBreakpointSupportInfo()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setBreakpointSupportInfo (
            const std::string & supported ) [inline]
```

Set the `breakpointSupport` field.

Parameters

supported	The breakpointSupport field of the RegisterInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.11 setCanonicalRn()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setCanonicalRn (
            uint64_t canonicalRn_ )  [inline]
```

Set the `canonicalRn` field.
Note: Use setCanonicalRnElfDwarf() when using the "ElfDwarf" scheme.

Parameters

canonicalRn	The canonicalRn field of the RegisterInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.12 setCanonicalRnElfDwarf()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setCanonicalRnElfDwarf (
            uint16_t architecture,
            uint16_t dwarfRegNum )  [inline]
```

Set the `canonicalRn` field for "ElfDwarf" scheme.

Parameters

architecture	ELF EM_* constant for architecture.
dwarfRegNum	DWARF register number for architecture.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.13 setCname()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setCname (
            const std::string & cname )  [inline]
```

Set the `cname` field.

Parameters

cname	The cname field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.14 setDescription()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setDescription (
            const std::string & description ) [inline]
```

Set the `description` field.

Parameters

description	The description field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.15 setFormat()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setFormat (
            const std::string & format ) [inline]
```

Set the `format` field.

Parameters

format	The format field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.16 setLsbOffset()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setLsbOffset (
            uint64_t lsbOffset ) [inline]
```

Set the `lsbOffset` field.

Parameters

lsbOffset	The lsbOffset field of the RegisterInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.17 setName()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setName (
            const std::string & name ) [inline]
```

Set the `name` field.

Parameters

name	The name field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.18 setParentRscId()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setParentRscId (
            ResourceId parentRscId ) [inline]
```

Set the `parentRscId` field.
This function makes this register a child of the specified parent. It is not necessary to call this function when adding child registers using the addField() function.

Parameters

parent↩ RscId	The rscId of the parent register.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.19 setReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setReadDelegate ( ) [inline]
```

Set the delegate to read the resource.
Set a delegate which calls function FUNC().
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Template Parameters

FUNC	A resource read delegate function.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.20 setReadDelegate() [2/3]

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setReadDelegate (
            ResourceReadDelegate readDelegate ) [inline]
```

Set the delegate to read the resource.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Parameters

readDelegate	ResourceReadDelegate object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.21 setReadDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setReadDelegate (
            T * instance )  [inline]
```

Set the delegate to read the resource.
Set a delegate which calls METHOD() on an instance of class T.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Template Parameters

T	A class that defines a method with the right signature to be a resource read delegate.
METHOD	A resource read delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.22 setResetData() [1/2]

```
template<typename T >
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setResetData (
            std::initializer_list< T > && t )  [inline]
```

Set the `resetData` field for wide registers.
This function accepts a braced initializer-list and is otherwise idential to setResetDataFromContainer().
Each element will be promoted/narrowed to uint64_t.

Parameters

t	Braced initializer-list.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.23 setResetData() [2/2]

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setResetData (
            uint64_t value ) [inline]
```

Set the `resetData` field to a value <= 64 bit.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

value	resetData value of the register.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.24 setResetDataFromContainer()

```
template<typename Container >
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setResetDataFromContainer (
            const Container & container ) [inline]
```

Set the `resetData` field for wide registers.
Container must be a type which allows to iterate over uint64_t bit chunks of the value,
least significant bits first, for example std::array<uint64_t> or std::vector<uint64_t>.
Each element of the container will be promoted/narrowed to uint64_t.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

container	Container containing the value in 64-bit chunks.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.25 setResetString()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setResetString (
            const std::string & resetString ) [inline]
```

Set the `resetString` field.
Set the reset value for string registers.

Parameters

resetString	The resetString field of the RegisterInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.26 setRwMode()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setRwMode (
            const std::string & rwMode )  [inline]
```

Set the `rwMode` field.

Parameters

rwMode	The rwMode field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.27 setSubRscId()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setSubRscId (
            uint64_t subRscId )  [inline]
```

Set the `subRscId` field.

Parameters

subRscId	The subRscId field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.28 setTag() [1/2]

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setTag (
            const std::string & tag )  [inline]
```

Set the named boolean tag to true (e.g. isPc)

Parameters

tag	The name of the tag to set.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.29 setTag() [2/2]

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setTag (
            const std::string & tag,
            const IrisValue & value )  [inline]
```

Set a tag to the specified value.

Parameters

tag	The name of the tag to set.
value	The value to set the tag to.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.30 setType()

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setType (
            const std::string & type )  [inline]
```

Set the `type` field.

Parameters

type	The type field of the ResourceInfo object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.31 setWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setWriteDelegate ( )  [inline]
```

Set the delegate to write the resource.
Set a delegate which calls function FUNC().
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Template Parameters

FUNC	A resource write delegate function.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.32 setWriteDelegate() [2/3]

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setWriteDelegate (
            ResourceWriteDelegate writeDelegate )  [inline]
```

Set the delegate to write the resource.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Parameters

writeDelegate	ResourceWriteDelegate object.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.33 setWriteDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &)
METHOD>
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setWriteDelegate (
            T * instance )  [inline]
```

Set the delegate to write the resource.
Set a delegate which calls METHOD() on an instance of class T.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Template Parameters

T	A class that defines a method with the right signature to be a resource write delegate.
METHOD	A resource write delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.34 setWriteMask() [1/2]

```
template<typename T >
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setWriteMask (
            std::initializer_list< T > && t )  [inline]
```

Set the `writeMask` field for wide registers.
This function accepts a braced initializer-list and is otherwise idential to setWriteMaskFromContainer().
Each element will be promoted/narrowed to uint64_t.

Parameters

t	Braced initializer-list.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.35 setWriteMask() [2/2]

```
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setWriteMask (
            uint64_t value ) [inline]
```

Set the `writeMask` field to a value <= 64 bit.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

value	writeMask value of the register.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

8.8.2.36 setWriteMaskFromContainer()

```
template<typename Container >
FieldBuilder & iris::IrisInstanceBuilder::FieldBuilder::setWriteMaskFromContainer (
            const Container & container ) [inline]
```

Set the `writeMask` field for wide registers.
Container must be a type which allows to iterate over uint64_t bit chunks of the value,
least significant bits first, for example std::array<uint64_t> or std::vector<uint64_t>.
Each element of the container will be promoted/narrowed to uint64_t.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

container	Container containing the value in 64-bit chunks.

Returns

A reference to this FieldBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.9 iris::GetDisassemblyArgs Struct Reference

Public Attributes

- uint64_t **address**
- AttributeValueMap **attrib**
- uint64_t **count**
- std::vector< DisassemblyLine > & **disassemblyLineOut**
- uint64_t **maxAddr**
- std::string **mode**
- MemorySpaceId **spaceId**

The documentation for this struct was generated from the following file:

- IrisInstanceDisassembler.h

8.10 iris::IrisCConnection Class Reference

Provide an IrisConnectionInterface which loads an IrisC library.
`#include <IrisCConnection.h>`
Inherits IrisConnectionInterface.

Public Member Functions

- virtual IrisInterface * **getIrisInterface** () override

  Get the IrisInterface for this connection. See also IrisConnectionInterface::getIrisInterface().
- **IrisCConnection** (IrisC_Functions *functions)
- virtual IrisErrorCode **processAsyncMessages** (bool waitForAMessage) override

  Process asynchronous messages for the calling thread. See also IrisConnectionInterface::processAsyncMessages().
- virtual uint64_t **registerIrisInterfaceChannel** (IrisInterface *iris_interface, const std::string &connectionInfo) override

  Register a communication channel. See also IrisConnectionInterface::registerIrisInterfaceChannel().
- virtual void **unregisterIrisInterfaceChannel** (uint64_t channelId) override

  Unregister a communication channel. See also IrisConnectionInterface::unregisterIrisInterfaceChannel().

Protected Member Functions

- int64_t **IrisC_handleMessage** (const uint64_t *message)

  Wrapper functions to call the underlying IrisC functions.
- int64_t **IrisC_processAsyncMessages** (bool waitForAMessage)
- int64_t **IrisC_registerChannel** (IrisC_CommunicationChannel *channel, uint64_t *channel_id_out)
- int64_t **IrisC_unregisterChannel** (uint64_t channel_id)
- **IrisCConnection** ()

  Construct an empty object. Used by subclasses that need to load a DSO and call init().

Protected Attributes

- void * **iris_c_context**

  Context pointer to use when calling IrisC_* functions. This is also needed by subclasses.

8.10.1 Detailed Description

Provide an IrisConnectionInterface which loads an IrisC library.

See also

IrisClient

IrisGlobalInstance

The documentation for this class was generated from the following file:

- IrisCConnection.h

8.11 iris::IrisClient Class Reference

Inherits IrisInterface, impl::IrisProcessEventsInterface, IrisConnectionInterface, and iris::IrisInstance.

Public Member Functions

- void connect (const std::string &connectionSpec)
- IrisErrorCode connect (const std::string &hostname, uint16_t port, unsigned timeoutInMs, std::string &error↩ ResponseOut)
- void connectCommandLine (const std::vector< std::string > &commandLine_, const std::string &program↩ Name)
- void **connectCommandLineKeepOtherArgs** (std::vector< std::string > &commandLine, const std::string &programName)

  Same as connectCommandLine() but remove all known arguments from commandLine and keep all other arguments in commandLine.
- void connectSocketFd (SocketFd socketfd, unsigned timeoutInMs=1000)
- IrisErrorCode disconnect ()
- bool disconnectAndWaitForChildToExit (double timeoutInMs=5000, double timeoutInMsAfterSigInt=5000, double timeoutInMsAfterSigKill=5000)
- uint64_t **getChildPid** () const

  Get child process id of previously spawned process or 0 if no process was spawned yet using spawnAndConnect().
- std::string **getConnectionStr** () const

  Get connection string, describing the Iris server we are connected to.
- impl::IrisRpcAdapterTcp::Format **getEffectiveSendingFormat** () const

  Get effective sending format that Rpc adapter uses.
- IrisInstance & getIrisInstance ()
- virtual IrisInterface * **getIrisInterface** () override
- int **getLastExitStatus** () const

  Get last exit status of child process, or -1 if the child process did not yet exit.
- IrisInterface * **getSendingInterface** ()

  Get interface for sending messages to the server.
- void initServiceServer (impl::IrisTcpSocket *socket_)
- **IrisClient** (const service::IrisServiceTcpServer *, const std::string &instName=std::string())

  Service constructor to initialize IrisService Server on IrisService side.
- IrisClient (const std::string &hostname, uint16_t port, const std::string &instName=std::string())

  Construct a connection to an Iris server.
- IrisClient (const std::string &instName, const std::vector< std::string > &commandLine_, const std::string &programName)
- **IrisClient** (const std::string &instName=std::string(), const std::string &connectionSpec=std::string())

  Connect accordimng to connectionSpec. See connectionHelpStr for syntax and semantics.
- bool **isConnected** () const

  Return true iff connected to a server.
- void loadPlugin (const std::string &plugin_name)
- virtual IrisErrorCode **processAsyncMessages** (bool waitForAMessage) override
- virtual void processEvents () override
- uint64_t **registerChannel** (IrisC_CommunicationChannel *channel, const ::std::string &connectionInfo)
- virtual uint64_t **registerIrisInterfaceChannel** (IrisInterface *iris_interface, const std::string &connectionInfo) override
- void setInstanceName (const std::string &instName)
- void **setIrisMessageLogLevel** (unsigned level, bool increaseOnly=false)

  Enable message logging.
- void **setPreferredSendingFormat** (impl::IrisRpcAdapterTcp::Format p)

  Set preferred sending format that Rpc adapter uses.
- void setSleepOnDestructionMs (uint64_t sleepOnDestructionMs_)
- void **setVerbose** (unsigned level, bool increaseOnly=false)

  Set verbose level.
- void spawnAndConnect (const std::vector< std::string > &modelCommandLine, const std::string &additionalServerArgs=std::string(), const std::string &additionalClientArgs=std::string())

- virtual void stopWaitForEvent () override
- void **unloadPlugin** ()
- void **unregisterChannel** (uint64_t channelId)
- virtual void **unregisterIrisInterfaceChannel** (uint64_t channelId) override
- virtual void waitForEvent () override
- bool waitpidWithTimeout (uint64_t pid, int *status, int options, double timeoutInMs)
- virtual ~**IrisClient** ()

  Destructor.

Static Public Member Functions

- static std::string getConnectCommandLineHelp ()

Public Attributes

- const std::string connectionHelpStr

  Connection help string.

Additional Inherited Members

8.11.1 Constructor & Destructor Documentation

8.11.1.1 IrisClient() [1/2]

```
iris::IrisClient::IrisClient (
            const std::string & instName,
            const std::vector< std::string > & commandLine_,
            const std::string & programName )  [inline]
```

Connect via command-line-like interface. See connectCommandLine().

8.11.1.2 IrisClient() [2/2]

```
iris::IrisClient::IrisClient (
            const std::string & hostname,
            uint16_t port,
            const std::string & instName = std::string() )  [inline]
```

Construct a connection to an Iris server.

Parameters

hostname	Hostname of the Iris server. This can be an IP address. For example: • "192.168.0.5" IP address of a different host. • "127.0.0.1" Loopback IP address to connect to a server on the same machine. • "localhost" Hostname of the loopback interface. Port == 0 means to scan ports 7100 to 7109. • "foo.bar.com" Hostname of a remote machine.
port	Server port number to connect to on the host.

8.11.2 Member Function Documentation

8.11.2.1 connect() [1/2]

```
void iris::IrisClient::connect (
            const std::string & connectionSpec )  [inline]
```

Connect to an Iris server.
The connection details are specified as a string. See "connectionHelpStr" for syntax. This function is self documenting: Passing "help" will return a list of all supported connection types and their syntax, as an E_help_↩ message error.
This throws E_not_connected when connectionSpec was erroneous, and E_socket_error or E_connection_refused when the connection could not be established. In case of an error the socket is closed.

8.11.2.2 connect() [2/2]

```
IrisErrorCode iris::IrisClient::connect (
            const std::string & hostname,
            uint16_t port,
            unsigned timeoutInMs,
            std::string & errorResponseOut )  [inline]
```

Connect to TCP server on hostname:port.
If hostname == "localhost" and port == 0 then a port scan on ports 7100 to 7109 is done. In case of an error the socket is closed.

8.11.2.3 connectCommandLine()

```
void iris::IrisClient::connectCommandLine (
            const std::vector< std::string > & commandLine_,
            const std::string & programName )  [inline]
```

Connect via command-line-like interface.
This high-level function is convenient for tools which transparently want to support all connection types.
This either starts a model child process ("--") or connects to a running model process ("tcp", default). See getConnectCommandLineHelp() for command line syntax and supported arguments. All errors are reported via exceptions.
This is a frontend to spawnAndConnect() and to connect(hostname, port).
commandLine: See getConnectCommandLineHelp() (or pass the command line ["help"]) for supported arguments.
programName: This is just used in the help message.

8.11.2.4 connectSocketFd()

```
void iris::IrisClient::connectSocketFd (
            SocketFd socketfd,
            unsigned timeoutInMs = 1000 )  [inline]
```

Connect using an existing socketFd. All errors are reported by exceptions. In case of an error the socket is closed.

8.11.2.5 disconnect()

```
IrisErrorCode iris::IrisClient::disconnect ( )  [inline]
```

Disconnect from server. Close socket. (Only for mode IRIS_TCP_CLIENT.)

8.11.2.6 disconnectAndWaitForChildToExit()

```
bool iris::IrisClient::disconnectAndWaitForChildToExit (
            double timeoutInMs = 5000,
            double timeoutInMsAfterSigInt = 5000,
            double timeoutInMsAfterSigKill = 5000 )  [inline]
```

Disconnect and wait for child process (previously spawned with spawnAndConnect()) to exit. If no model was spawned this is silently ignored.
Wait at most timeoutInMs until the child exits. If the child did not exit by then, send a SIGINT and wait for timeout↩ InMsAfterSigInt until the child exits. If the child did not exit by then, send a SIGKILL and wait for timeoutInMsAfter↩ SigKill until the child exits. If the child did not exit by then, an E_not_connected exception is thrown. If timeoutInMs is

0, do not wait and continue with SIGINT. If timeoutAfterSigInt is 0, do not issue a SIGINT and continue with SIGKILL If timeoutAfterSigKill is 0, do not issue a SIGKILL and throw an E_not_connected exception. If any of the timeouts is < 0, wait indefinitely.
Return true if the child exited, else false.

8.11.2.7 getConnectCommandLineHelp()

```
static std::string iris::IrisClient::getConnectCommandLineHelp ( )  [inline], [static]
```

Get help string for connectCommandLine(). This can be used by tools using connectCommandLine() as part of their –help message.

8.11.2.8 getIrisInstance()

```
IrisInstance & iris::IrisClient::getIrisInstance ( )  [inline]
```

Get IrisInstance. This is here just for backward compatibility when IrisClient was not an IrisInstance but contained one.

8.11.2.9 initServiceServer()

```
void iris::IrisClient::initServiceServer (
            impl::IrisTcpSocket * socket_ )  [inline]
```

Initialize as an IrisService server, only used in IRIS_SERVICE_SERVER mode. This function will store pointer to IrisTcpSocket created by IrisService and initialize adapter as a server. -socket_ pointer to IrisTcpSocket created by IrisService when receiving new connection. -return Nothing.

8.11.2.10 loadPlugin()

```
void iris::IrisClient::loadPlugin (
            const std::string & plugin_name )  [inline]
```

Load Plugin function, only used in IRIS_SERVICE_SERVER mode Only one plugin can be loaded at a a time

8.11.2.11 processEvents()

```
virtual void iris::IrisClient::processEvents ( )  [inline], [override], [virtual]
```

Client main processing function.

- Check for incoming requests/responses and process them .
- Check for pending outgoing requests/responses and process them. This function is ideal for integrating the client into other processing environments in one of the following ways: (1) Thread-less: Requests are only executed from within processEvents().
- pro: Iris request and responses are always synchronized with the rest of the code of the client. No explicit synchronization (mutexes etc.) necessary.
- con: No blocking Iris requests can be called from within received synchronous callbacks. (2) Asynchronous (handleRequestAsynchronously = true): Requests are executed in another thread
- pro: Blocking Iris requests can be called from within received synchronous callbacks transparently.
- con: Received Iris requests are called on another thread and they require explicit synchronization to be synchronized with the rest of the code of the client. It is harmless to call this function when there is nothing to do.

8.11.2.12 setInstanceName()

```
void iris::IrisClient::setInstanceName (
            const std::string & instName )  [inline]
```

Set instance name of the contained Iris instance returned by getIrisInstance. This must be called before connect().

8.11.2.13 setSleepOnDestructionMs()

```
void iris::IrisClient::setSleepOnDestructionMs (
            uint64_t sleepOnDestructionMs_ )  [inline]
```

Sleep a short time on destruction to de-interleave output by different processes. This has no functional impact or purpose. It just beautifies the output on stdout.

8.11.2.14 spawnAndConnect()

```
void iris::IrisClient::spawnAndConnect (
            const std::vector< std::string > & modelCommandLine,
            const std::string & additionalServerArgs = std::string(),
            const std::string & additionalClientArgs = std::string() )  [inline]
```

Spawn model and connect to it. All errors are reported via exceptions. additionalServerArgs are added to the models –iris-connect argument and ultimately passed to IrisTcpServer::startServer(), for example "verbose=1" to enable verbose messages. additionalClientArgs are added to the argument passed to IrisClient::connect(), for example "verbose=1,timeout=2000" to enable verbose messages and a 2 second timeout.

8.11.2.15 stopWaitForEvent()

```
virtual void iris::IrisClient::stopWaitForEvent ( )  [inline], [override], [virtual]
```

Stop waiting in waitForEvent(). Return from waitForEvent() as soon as possible even without a socket event.

8.11.2.16 waitForEvent()

```
virtual void iris::IrisClient::waitForEvent ( )  [inline], [override], [virtual]
```

Wait for any event which would cause processEvents() to do some work. This function intentionally blocks until there is something useful to do. This function can be interrupted by calling stopWaitForEvent().

8.11.2.17 waitpidWithTimeout()

```
bool iris::IrisClient::waitpidWithTimeout (
            uint64_t pid,
            int * status,
            int options,
            double timeoutInMs )  [inline]
```

waitpid() with timeout. Throw exceptions on errors. Return true if the child exited within the timeout, else false.

8.11.3 Member Data Documentation

8.11.3.1 connectionHelpStr

```
const std::string iris::IrisClient::connectionHelpStr
```

Initial value:

```
=
        "Supported connection types:\n"
        "tcp[=HOST][,port=PORT][,timeout=T]\n"
        "  Connect to an Iris TCP server on HOST:PORT.\n"
        "  The default for HOST is 'localhost' and the default for PORT is 0 if HOST is 'localhost' and 7100
       otherwise. If PORT is 0 then a port scan on ports 7100 to 7109 is done.\n"
        "  T is the connection timeout in ms (defaults to 100 if PORT==0, else 1000).\n"
        "\n"
        "socketfd=FD[,timeout=T]\n"
        "  Use socket file descriptor FD as an established UNIX domain socket connection.\n"
        "  T is the timeout for the Iris handshake in ms.\n"
        "\n"
        "General parameters:\n"
        "  verbose[=N]: Increase verbose level of IrisClient to level N (0..3).\n"
        "  iris-log[=N]: Log Iris functions calls (1=pretty, 2=JSON, 3=JSON-multiline, +8=U64JSON, +16=time,
       +32=reltime).\n"
```

Connection help string.

The documentation for this class was generated from the following file:

- IrisClient.h

8.12 iris::IrisCommandLineParser Class Reference

`#include <IrisCommandLineParser.h>`

Classes

- struct Option

  Option container.

Public Member Functions

- Option & addOption (char shortOption, const std::string &longOption, const std::string &help, const std::string &formalArgumentName, int64_t defaultValue)
- Option & addOption (char shortOption, const std::string &longOption, const std::string &help, const std::string &formalArgumentName=std::string(), const std::string &defaultValue=std::string())
- void clear ()
- double getDbl (const std::string &longOption) const
- std::string getHelpMessage () const
- int64_t getInt (const std::string &longOption) const
- std::vector< std::string > **getList** (const std::string &longOption) const

  Get list of elements of a list option.
- std::map< std::string, std::string > getMap (const std::string &longOption) const
- std::vector< std::string > & getNonOptionArguments ()
- const std::vector< std::string > & **getNonOptionArguments** () const

  Get non-option arguments.
- std::string **getProgramName** () const

  Get program name.
- std::string **getStr** (const std::string &longOption) const

  Get string value.
- uint64_t **getSwitch** (const std::string &longOption) const

  Check how many times an option switch (an option without an argument) was specified.
- uint64_t getUint (const std::string &longOption) const
- IrisCommandLineParser (const std::string &programName, const std::string &usageHeader, const std::string &versionStr, bool keepDashDash=false)
- bool isSpecified (const std::string &longOption) const
- void noNonOptionArguments ()
- bool **operator()** (const std::string &longOption) const

  Check whether an option was specified.
- int **parseCommandLine** (int argc, char **argv)
- int parseCommandLine (int argc, const char **argv)
- void pleaseSpecifyOneOf (const std::vector< std::string > &options, const std::vector< std::string > &formalNonOptionArguments=std::vector< std::string >())
- int **printError** (const std::string &message) const

  Print error message (and do not exit).
- int printErrorAndExit (const std::exception &e) const
- int printErrorAndExit (const std::string &message) const
- int printMessage (const std::string &message, int error=0, bool exit=false) const
- void **setHelpMessagePad** (uint64_t pad)

  Set help message starting position.
- void setMessageFunc (const std::function< int(const std::string &message, int error, bool exit)> &messageFunc)
- void **setProgramName** (const std::string &programName_, bool append=false)

  Set/override program name.
- void setValue (const std::string &longOption, const std::string &value, bool append=false)
- void throwError (const std::string &message) const
- void unsetValue (const std::string &longOption)

Static Public Member Functions

- static int defaultMessageFunc (const std::string &message, int error, bool exit)

Static Public Attributes

- static const bool **KeepDashDash** = true

  Keep "--" in the non-option arguments because it has semantics for the application beyond stopping option parsing.

8.12.1 Detailed Description

Generic command line parser.
This covers roughly all features supported by GNU getopt_long() and provides -h/–help and –version.
Usage:

1. Declare options by calling addOption() for each option.
2. Parse command line by calling parseCommandLine().
3. Retrieve command line option values by calling the get...() functions.

Example:

8.12.2 Constructor & Destructor Documentation

8.12.2.1 IrisCommandLineParser()

```
iris::IrisCommandLineParser::IrisCommandLineParser (
            const std::string & programName,
            const std::string & usageHeader,
            const std::string & versionStr,
            bool keepDashDash = false )
```

Constructor. programName, usageHeader and versionStr: Appears in the –help and –version messages. keep↩ DashDash: Keep "--" in the non-option arguments because it has semantics for the application beyond stopping option parsing.

8.12.3 Member Function Documentation

8.12.3.1 addOption() [1/2]

```
Option & iris::IrisCommandLineParser::addOption (
            char shortOption,
            const std::string & longOption,
            const std::string & help,
            const std::string & formalArgumentName,
            int64_t defaultValue )  [inline]
```

Same as above for integer defaults. (Without this overload, specifying an integer default of 0 will automatically get converted to a NULL const char∗ and then to a std::string which segfaults.)

8.12.3.2 addOption() [2/2]

```
Option & iris::IrisCommandLineParser::addOption (
            char shortOption,
            const std::string & longOption,
            const std::string & help,
            const std::string & formalArgumentName = std::string(),
            const std::string & defaultValue = std::string() )
```

Add command line option. shortOption: Single character or 0 if no short option. longOption: Long option (mandatory, must be unique and non-empty). help: Description for –help. formalArgumentName: Empty means: This option has no argument (switch). Nonempty means: This option has an argument and this is named 'formalArgument↩ Name' in the –help message. defaultValue: Default value of this option when not specified on the command line. When defaultValue is not specified: By default getSwitch(), getInt() and getUint() return 0 and getStr() returns an empty string.

8.12.3.3 clear()

```
void iris::IrisCommandLineParser::clear ( )
```

Clear all values parsed by a previous parseCommandLine call. All options will be reset to their default values. All option definitions (addOption()) will be preserved.

8.12.3.4 defaultMessageFunc()

```
static int iris::IrisCommandLineParser::defaultMessageFunc (
            const std::string & message,
            int error,
            bool exit ) [static]
```

Default message function. The default message function prints message on stdout and exits with "error" status if exit==true, else it returns error status.

8.12.3.5 getDbl()

```
double iris::IrisCommandLineParser::getDbl (
            const std::string & longOption ) const
```

Get double value. (This will print an error and exit when there is a parse error.)

8.12.3.6 getHelpMessage()

```
std::string iris::IrisCommandLineParser::getHelpMessage ( ) const
```

Get help message. (parserCommandLine() automatically prints this on –help so there is usually no need to call this function.)

8.12.3.7 getInt()

```
int64_t iris::IrisCommandLineParser::getInt (
            const std::string & longOption ) const
```

Get integer value. (This will print an error and exit when there is a parse error.)

8.12.3.8 getMap()

```
std::map< std::string, std::string > iris::IrisCommandLineParser::getMap (
            const std::string & longOption ) const
```

Get NAME->VALUE map of elements of a list option. The elements are assumed to have the format "NAME=↩ VALUE" or "NAME". If "=VALUE" is missing then VALUE is the empty string.

8.12.3.9 getNonOptionArguments()

```
std::vector< std::string > & iris::IrisCommandLineParser::getNonOptionArguments ( ) [inline]
```

Get read/write access to non-option arguments. This is useful when chaining different non-option argument parsers.

8.12.3.10 getUint()

```
uint64_t iris::IrisCommandLineParser::getUint (
            const std::string & longOption ) const
```

Get unsigned integer value. (This will print an error and exit when there is a parse error.)

8.12.3.11 isSpecified()

```
bool iris::IrisCommandLineParser::isSpecified (
            const std::string & longOption ) const
```

Return true iff option is specified explicitly on the command line. (This can be used to detect whether an option was present on the command line even if it was just set to its default value.)

8.12.3.12 noNonOptionArguments()

```
void iris::IrisCommandLineParser::noNonOptionArguments ( )
```

Print an error for each non-option argument and exit if any non-option arguments are present. Call this after parseCommandLine() for programs which do not support any non-option arguments as these are otherwise silently ignored.

8.12.3.13 parseCommandLine()

```
int iris::IrisCommandLineParser::parseCommandLine (
            int argc,
            const char ** argv )
```

Parse command line. After calling this function the named argument values can be retrieved by the get...() functions. All arguments after the first occurrence of a "--" argument are treated as non-option arguments. Also handles –help and –version and exit()s when these are specified.
argv[0] is ignored. The program name is passed in the constructor argument.
Calling parseCommandLine() again will ad and/or override options as if they were in a single command line.
Return value: By default parseCommandLine() exits (and so does not return) when it detects an error or when –help or –version was specified, so the return value can safely (and should) be ignored.
When the exit behavior is overridden by calling setMessageFunc() with a non-exiting function, then parseCommandLine() returns the return value of the message function or 0 when the message function was not called (no error and no –help/–version).
Note that parse errors in integers or doubles are only identified by the respective get∗() functions.

8.12.3.14 pleaseSpecifyOneOf()

```
void iris::IrisCommandLineParser::pleaseSpecifyOneOf (
            const std::vector< std::string > & options,
            const std::vector< std::string > & formalNonOptionArguments = std::vector< std↩
::string >() )
```

Check whether at least one of the options or non-option-arguments are specified and exit with an error message if not. Call this for programs which require at least one of these options or arguments to be set. If formalNonOption↩
Arguments is empty only options are checked.

8.12.3.15 printErrorAndExit() [1/2]

```
int iris::IrisCommandLineParser::printErrorAndExit (
            const std::exception & e ) const
```

Print error message and exit. Note that custom message functions may decide not to exit even on errors. In this case parseCommandLine() returns the return value of the message function.

8.12.3.16 printErrorAndExit() [2/2]

```
int iris::IrisCommandLineParser::printErrorAndExit (
            const std::string & message ) const
```

Print error message and exit. Note that custom message functions may decide not to exit even on errors. In this case parseCommandLine() returns the return value of the message function.

8.12.3.17 printMessage()

```
int iris::IrisCommandLineParser::printMessage (
            const std::string & message,
```

```
            int error = 0,
            bool exit = false ) const
```

Print message. This can be used by additional checks on the arguments to print warnings. This calls the message function set by setMessageFunc() or the defaultMessageFunc().

8.12.3.18 setMessageFunc()

```
void iris::IrisCommandLineParser::setMessageFunc (
            const std::function< int(const std::string &message, int error, bool exit)> &
messageFunc )
```

Set custom message function which prints errors (error!=0), –help and –version messages (error==0) and which potentially also exit()s (exit==true).
The default message function prints message on stdout and exits with "error" status if exit==true, else it returns error status.
Custom message functions may either exit, or they may return a value which is then returned by parserCommand↩ Line() for errors raised by parseCommandLine(). For errors in the get∗() functions the return value is ignored.

8.12.3.19 setValue()

```
void iris::IrisCommandLineParser::setValue (
            const std::string & longOption,
            const std::string & value,
            bool append = false )
```

Set/override command line option. By default overwrite the entire list for list options. Set append=true for list options to append to list.

8.12.3.20 throwError()

```
void iris::IrisCommandLineParser::throwError (
            const std::string & message ) const  [inline]
```

Throw E_error_message error. This is useful to print fatal errors from main and let the try/catch block do any cleanup (e.g. terminating child processes).

8.12.3.21 unsetValue()

```
void iris::IrisCommandLineParser::unsetValue (
            const std::string & longOption )
```

Unset command line option. Set value to default value and mark as not specified.
The documentation for this class was generated from the following file:

- IrisCommandLineParser.h

8.13 iris::IrisEventEmitter< ARGS > Class Template Reference

A helper class for generating Iris events.
`#include <IrisEventEmitter.h>`
Inherits IrisEventEmitterBase.

Public Member Functions

- **IrisEventEmitter** ()

  Construct an event emitter.
- void operator() (ARGS... args)

  Emit an event.

8.13.1 Detailed Description

template<typename... ARGS>
class iris::IrisEventEmitter< ARGS >

A helper class for generating Iris events.

Template Parameters

ARGS	Argument types corresponding to the fields in this event.

Use IrisEventEmitter with IrisInstanceBuilder to add events to your Iris instance:

```
// Declare an event emitter
iris::IrisEventEmitter<uint64_t, bool> my_event;
// Add it to an Iris instance
iris::IrisInstance my_instance(...);
my_instance->getBuilder()->addEventSource("MY_EVENT", my_event)
    .addField("FOO",  "uint", 8, "A value")
    .addField("FLAG", "bool", 1, "A flag");
// Emit an event
my_event(0x1234, true);
```

8.13.2 Member Function Documentation

8.13.2.1 operator()()

```
template<typename...  ARGS>
void iris::IrisEventEmitter< ARGS >::operator() (
            ARGS...  args )  [inline]
```

Emit an event.
The arguments to this function are the fields of the event source, in the same order that they appear in the template arguments to the IrisEventEmitter class.
The documentation for this class was generated from the following file:

- IrisEventEmitter.h

8.14 iris::IrisEventRegistry Class Reference

Class to register Iris event streams for an event.
`#include <IrisInstanceEvent.h>`

Public Types

- typedef std::set< EventStream * >::const_iterator **iterator**

Public Member Functions

- template<class T >
  void addField (const IrisU64StringConstant &field, const T &value) const

  Add a field value.
- template<class T >
  void addFieldSlow (const std::string &field, const T &value) const

  Add a field value.
- iterator begin () const

  Get an iterator to the beginning of the event stream set.
- void **emitEventBegin** (uint64_t time, uint64_t pc=IRIS_UINT64_MAX) const
- void emitEventEnd () const

  Emit the callback.

- bool empty () const

  Return true if no event streams are registered.
- iterator end () const

  Get an iterator to the end of the event stream set.
- template<class T , typename F >
  void forEach (F &&func) const

  Call a function for each event stream.
- bool registerEventStream (EventStream *evStream)

  Register an event stream.
- bool unregisterEventStream (EventStream *evStream)

  Unregister an event stream.

8.14.1 Detailed Description

Class to register Iris event streams for an event.

8.14.2 Member Function Documentation

8.14.2.1 addField()

```
template<class T >
void iris::IrisEventRegistry::addField (
            const IrisU64StringConstant & field,
            const T & value ) const  [inline]
```

Add a field value.
This is supported for all types supported by IrisU64JsonWriter and IrisObjects.h. Fast variant for argument names up to 23 chars. Use this if you can.

Template Parameters

T	The type of `value`.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.14.2.2 addFieldSlow()

```
template<class T >
void iris::IrisEventRegistry::addFieldSlow (
            const std::string & field,
            const T & value ) const  [inline]
```

Add a field value.
This is supported for all types supported by IrisU64JsonWriter and IrisObjects.h. Slow variant for argument names with more than 23 chars. Do not use unless you have to.

Template Parameters

T	The type of `value`.

Parameters

field	The name of the field whose value is set.
value	The value of the field.

8.14.2.3 begin()

```
iterator iris::IrisEventRegistry::begin ( ) const  [inline]
```

Get an iterator to the beginning of the event stream set.

See also

end

Returns

An iterator to the beginning of the event stream set.

8.14.2.4 emitEventEnd()

```
void iris::IrisEventRegistry::emitEventEnd ( ) const
```

Emit the callback.
This also checks the ranges and maintains the counter.

8.14.2.5 empty()

```
bool iris::IrisEventRegistry::empty ( ) const  [inline]
```

Return true if no event streams are registered.

Returns

`true` if no event streams are registered.

8.14.2.6 end()

```
iterator iris::IrisEventRegistry::end ( ) const  [inline]
```

Get an iterator to the end of the event stream set.

See also

begin

Returns

An iterator to the end of the event stream set.

8.14.2.7 forEach()

```
template<class T , typename F >
void iris::IrisEventRegistry::forEach (
            F && func ) const  [inline]
```

Call a function for each event stream.
This function can be used as an alternative to addField()/addFieldSlow(), when each event stream needs to be handled individually, for example because the event stream has options or because only selected fields should be emitted.

The main use-case of this function is to emit the fields of all event streams.
Example of an event source which optionally allows inverting its data: class MyEventStream : public iris::IrisEventStream {...} IrisEventRegistry evreg; In the callback set with (IrisInstanceBuilder.addSource().) set↩ EventStreamCreateDelegate() create a new event stream with new MyEventStream(evreg);
// Emit event. evreg.emitEventBegin(time, pc); // Start building the callback data. evreg.forEach<MyEvent↩ Stream>([&](MyEventStream& es) { es.addField(ISTR("DATA"), es.invert ? ~data : data); }); evreg.emitEventEnd(); // Emit the callback.

Template Parameters

T	Class derived from IrisEventStream.
F	Function to be called for each event stream (usually a lambda function).

8.14.2.8 registerEventStream()

```
bool iris::IrisEventRegistry::registerEventStream (
            EventStream * evStream )
```

Register an event stream.

Parameters

evStream	The stream to be registered.

Returns

`true` if the stream was registered successfully.

8.14.2.9 unregisterEventStream()

```
bool iris::IrisEventRegistry::unregisterEventStream (
            EventStream * evStream )
```

Unregister an event stream.

Parameters

evStream	The stream to be unregistered.

Returns

`true` if the stream was unregistered successfully.

The documentation for this class was generated from the following file:

- IrisInstanceEvent.h

8.15 iris::IrisEventStream Class Reference

Event stream class for Iris-specific events.
`#include <IrisInstanceEvent.h>`
Inherits iris::EventStream.

Public Member Functions

- virtual IrisErrorCode disable () override

Disable this event stream.

- virtual IrisErrorCode enable () override

  Enable this event stream.

- **IrisEventStream** (IrisEventRegistry *registry_)

Additional Inherited Members

8.15.1 Detailed Description

Event stream class for Iris-specific events.

8.15.2 Member Function Documentation

8.15.2.1 disable()

```
virtual IrisErrorCode iris::IrisEventStream::disable ( )  [override], [virtual]
```

Disable this event stream.
This function is only called when isEnabled()/enabled == true. It is not necessary to verify this inside the disable() method.

Returns

An error code indicating whether the event stream was successfully disabled. This should be E_ok if it was disabled or E_error_disabling_event_stream if it could not be disabled.

Implements iris::EventStream.

8.15.2.2 enable()

```
virtual IrisErrorCode iris::IrisEventStream::enable ( )  [override], [virtual]
```

Enable this event stream.
This function is only called when isEnabled()/enabled == false. It is not necessary to verify this inside the enable() method.

Returns

An error code indicating whether the event stream was successfully enabled. This should be E_ok if it was enabled or E_error_enabling_event_stream if it could not be enabled.

Implements iris::EventStream.
The documentation for this class was generated from the following file:

- IrisInstanceEvent.h

8.16 iris::IrisGlobalInstance Class Reference

Inherits IrisInterface, and IrisConnectionInterface.

Public Member Functions

- void **emitLogMessage** (const std::string &message, const std::string &severityLevel)
- IrisInstance & getIrisInstance ()
- virtual IrisInterface * **getIrisInterface** () override

  Get the IrisInterface for this connection.

- **IrisGlobalInstance** ()

  Constructor.

- virtual void **irisHandleMessage** (const uint64_t *message) override

Handle incoming Iris messages.

- virtual IrisErrorCode **processAsyncMessages** (bool waitForAMessage) override
- uint64_t registerChannel (IrisC_CommunicationChannel *channel, const std::string &connectionInfo)
- virtual uint64_t registerIrisInterfaceChannel (IrisInterface *iris_interface, const std::string &connectionInfo) override
- virtual void **setIrisProxyInterface** (IrisProxyInterface *irisProxyInterface_) override

  Set proxy interface.
- void **setLogLevel** (unsigned level)
- void setLogMessageFunction (std::function< IrisErrorCode(const std::string &, const std::string &)> func)

  Set the function which will be called to log message for logger_logMessage Iris API.
- void **unregisterChannel** (uint64_t channelId)

  Unregister a channel.
- virtual void unregisterIrisInterfaceChannel (uint64_t channelId) override
- ~**IrisGlobalInstance** ()

  Destructor.

8.16.1 Member Function Documentation

8.16.1.1 getIrisInstance()

```
IrisInstance & iris::IrisGlobalInstance::getIrisInstance ( )  [inline]
```

Get contained IrisInstance. This can be used as a generic client instance to call Iris functions.

8.16.1.2 registerChannel()

```
uint64_t iris::IrisGlobalInstance::registerChannel (
            IrisC_CommunicationChannel * channel,
            const std::string & connectionInfo )
```

Register a channel. Returns an associated channel id.

8.16.1.3 registerIrisInterfaceChannel()

```
virtual uint64_t iris::IrisGlobalInstance::registerIrisInterfaceChannel (
            IrisInterface * iris_interface,
            const std::string & connectionInfo )  [override], [virtual]
```

Register a local IrisInterface with the system. This allows it to receive messages (requests and responses). Returns the unique channelId used to identify this channel when registering instances.

8.16.1.4 setLogMessageFunction()

```
void iris::IrisGlobalInstance::setLogMessageFunction (
            std::function< IrisErrorCode(const std::string &, const std::string &)> func )
[inline]
```

Set the function which will be called to log message for logger_logMessage Iris API.

Parameters

func	A function object that will be called to log the message.

8.16.1.5 unregisterIrisInterfaceChannel()

```
virtual void iris::IrisGlobalInstance::unregisterIrisInterfaceChannel (
            uint64_t channelId )  [inline], [override], [virtual]
```

Unregister a previously registered channel. This will automatically unregister all instances associated with that channel.
The documentation for this class was generated from the following file:

- IrisGlobalInstance.h

8.17 iris::IrisInstance Class Reference

Inherited by iris::IrisClient.

Public Types

- using EventCallbackFunction = std::function< IrisErrorCode(EventStreamId, const IrisValueMap &, uint64_t, InstanceId, bool, std::string &)>

Public Member Functions

- void addCallback_IRIS_INSTANCE_REGISTRY_CHANGED (EventCallbackFunction f)
- void **clearCachedMetaInfo** ()

  Clear cached meta-information including the list of InstanceInfos for all instances in the system.
- void destroyAllEventStreams ()

  Destroy all event streams.
- void disableEvent (const std::string &eventSpec)

  Disable all matching event callback(s).
- void enableEvent (const std::string &eventSpec, std::function< void()> callback, bool syncEc=false)

  Enable event callback(s).
- void enableEvent (const std::string &eventSpec, std::function< void(const EventStreamInfo &eventStream↩ Info, IrisReceivedRequest &request)> callback, bool syncEc=ASYNCHRONOUS)

  Enable event callback(s).
- void eventBufferDestroyed (EventBufferId evBufId)

  Notify instance that a specific event buffer was just destroyed.
- std::vector< EventSourceInfo > findEventSources (const std::string &instancePathFilter="all")

  Find all event sources in the system.
- std::vector< EventStreamInfo > findEventSourcesAndFields (const std::string &spec, InstanceId default↩ InstId=IRIS_UINT64_MAX)

  Find specific event sources in the system.
- void **findEventSourcesAndFields** (const std::string &spec, std::vector< EventStreamInfo > &event↩ StreamInfosOut, InstanceId defaultInstId=IRIS_UINT64_MAX)
- std::vector< InstanceInfo > findInstanceInfos (const std::string &instancePathFilter="all")

  Find instance infos of all instances in the system.
- IrisInstanceBuilder * getBuilder ()

  Get the IrisInstanceBuilder object for this instance. This can be used to set up metadata and callbacks for standard Iris functions.
- const std::vector< EventSourceInfo > & **getEventSourceInfosOfAllInstances** ()

  Find all event sources of all instances in the system.
- InstanceId getInstanceId (const std::string &instName)

  Get instance id for a specifid instance name.
- InstanceInfo getInstanceInfo (const std::string &instancePathFilter)

  Get instance info of a specific instance in the system.
- const InstanceInfo & getInstanceInfo (InstanceId instId)

  Get InstanceInfo including properties for a specific instId.
- const std::vector< InstanceInfo > & getInstanceList ()

  Get list of InstanceInfos of all instances in the system, including properties.

- const std::string & getInstanceName () const

  Get the instance name of this instance. This is valid after registerInstance() returns.
- std::string getInstanceName (InstanceId instId)

  Get instance name for a specifid instId.
- InstanceId getInstId () const

  Get the instance id of this instance. This is valid after registerInstance() returns.
- IrisInterface * getLocalIrisInterface ()

  Get the local IrisInterface of this instance. This is the interface that other instances use to send their requests and responses to this instance.
- IrisLogger & getLogger ()

  Get logger.
- MemorySpaceId getMemorySpaceId (InstanceId instId, const std::string &name)

  Get memory space id of memory space by name.
- MemorySpaceId **getMemorySpaceId** (InstanceId instId, uint64_t canonicalMsn)

  Get memory space id of memory space identified by its canonical memory space number (e.g. CanonicalMsnArm_* constant).
- const MemorySpaceInfo & getMemorySpaceInfo (InstanceId instId, const std::string &name)

  Get MemorySpaceInfo of memory space by name.
- const MemorySpaceInfo & **getMemorySpaceInfo** (InstanceId instId, uint64_t canonicalMsn)

  Get MemorySpaceInfo of memory space identified by its canonical memory space number (e.g. CanonicalMsnArm_* constant).
- const MemorySpaceInfo & **getMemorySpaceInfoById** (InstanceId instId, MemorySpaceId memorySpaceId)

  Get MemorySpaceInfo of memory space identified by its memory space id.
- const std::vector< MemorySpaceInfo > & **getMemorySpaceInfos** (InstanceId instId)

  Get list of MemorySpaceInfos.
- const PropertyMap & getPropertyMap () const

  Get property map.
- IrisInterface * getRemoteIrisInterface ()

  Get the remote Iris interface.
- const std::vector< ResourceGroupInfo > & **getResourceGroups** (InstanceId instId)

  Get list of resource groups.
- ResourceId getResourceId (InstanceId instId, const std::string &resourceSpec)

  Get resource id for a specific resource.
- const ResourceInfo & getResourceInfo (InstanceId instId, const std::string &resourceSpec)

  Get ResourceInfo for a specific resource.
- const ResourceInfo & **getResourceInfo** (InstanceId instId, ResourceId resourceId)

  Get ResourceInfo for a specific resource.
- const std::vector< ResourceInfo > & **getResourceInfos** (InstanceId instId)

  Get list of all resource infos of an instance.
- std::vector< ResourceInfo > getResourceInfos (InstanceId instId, const std::string &resourceSpec)

  Get zero or more matching resource infos of an instance.
- IrisCppAdapter & irisCall ()

  Get an IrisCppAdapter to call an Iris function of any other instance.
- IrisCppAdapter & irisCallNoThrow ()

  Get an IrisCppAdapter to call an Iris function of any other instance.
- IrisCppAdapter & irisCallThrow ()

  Get an IrisCppAdapter to call an Iris function of any other instance. When an Iris function returns an error response, this adapter always throws an exception. Usage:
- IrisInstance (IrisConnectionInterface *connection_interface=nullptr, const std::string &instName=std::string(), uint64_t flags=DEFAULT_FLAGS)

  Construct a new Iris instance.
- IrisInstance (IrisInstantiationContext *context)

Construct a new Iris instance using an IrisInstantiationContext.

- bool **isAdapterInitialized** () const
- bool isEventEnabled (const std::string &eventSpec)

  Check whether a certain event is already enabled through enableEvent.
- bool isRegistered () const
- bool isValidEvBufId (EventBufferId evBufId) const

  Check whether event buffer id is valid.
- void notifyStateChanged ()

  Notify client instances that the state of any resource/memory/table/disassembly etc changed.
- void **processAsyncRequests** ()

  Process async requests. Use this to keep the Iris system running while a thread is blocked waiting for something.
- template<class T >
  void publishCppInterface (const std::string &interfaceName, T *pointer, const std::string &jsonDescription)

  Publish a C++ interface XYZ through a new instance_getCppInterfaceXYZ() function.
- void registerEventBufferCallback (EventBufferCallbackDelegate delegate, const std::string &name, const std::string &description, const std::string &dlgInstanceTypeStr)

  Register an event buffer callback using an EventBufferCallbackDelegate.
- template<typename T , IrisErrorCode(T::*)(const EventBufferCallbackData &data) METHOD>
  void registerEventBufferCallback (T *instance, const std::string &name, const std::string &description, const std::string &dlgInstanceTypeStr)

  Register an event buffer callback using an EventBufferCallbackDelegate.
- template<class T >
  void registerEventBufferCallback (T *instance, const std::string &name, const std::string &description, void(T::*memberFunctionPtr)(IrisReceivedRequest &), const std::string &instanceTypeStr)

  Register an event buffer callback function.
- void registerEventCallback (EventCallbackDelegate delegate, const std::string &name, const std::string &description, const std::string &dlgInstanceTypeStr)

  Register a general event callback using an EventCallbackDelegate.
- template<typename T , IrisErrorCode(T::*)(uint64_t, const AttributeValueMap &, uint64_t, uint64_t, bool, std::string &) METHOD>
  void registerEventCallback (T *instance, const std::string &name, const std::string &description, const std::string &dlgInstanceTypeStr)

  Register a general event callback using an EventCallbackDelegate.
- template<class T >
  void registerEventCallback (T *instance, const std::string &name, const std::string &description, void(T::*memberFunctionPtr)(IrisReceivedRequest &), const std::string &instanceTypeStr)

  Register a general event callback.
- template<class T >
  void registerFunction (T *instance, const std::string &name, void(T::*memberFunctionPtr)(IrisReceivedRequest &), const std::string &functionInfoJson, const std::string &instanceTypeStr)

  Register an Iris function implementation.
- IrisErrorCode registerInstance (const std::string &instName, uint64_t flags=DEFAULT_FLAGS)

  Register this instance if it was not registered when constructed.
- uint64_t resourceRead (InstanceId instId, const std::string &resourceSpec)

  Read numeric resource and return its value.
- uint64_t resourceReadCrn (InstanceId instId, uint64_t canonicalRegisterNumber)

  Read numeric resource and return its value (using the canonical register number aka DWARF register id).
- std::string resourceReadStr (InstanceId instId, const std::string &resourceSpec)

  Read string resource, or read other resources as string.
- std::vector< uint64_t > resourceReadWide (InstanceId instId, const std::string &resourceSpec)

  Read wide numeric resource and return its value.
- void resourceWrite (InstanceId instId, const std::string &resourceSpec, const std::vector< uint64_t > &value)

  Write wide numeric resource.
- void resourceWrite (InstanceId instId, const std::string &resourceSpec, uint64_t value)

Write numeric resource.

- void resourceWriteCrn (InstanceId instId, uint64_t canonicalRegisterNumber, uint64_t value)

  Write numeric resource by canonical register number (aka DWARF register id).
- void resourceWriteStr (InstanceId instId, const std::string &resourceSpec, const std::string &value)

  Write string resource, or write numeric resource from string.
- bool sendRequest (IrisRequest &req)

  Send an Iris request or notification and potentially wait for a response.
- void sendResponse (const uint64_t *response)

  Send a response to the remote Iris interface.
- void **setAdapterInitialized** ()
- void setCallback_IRIS_SHUTDOWN_LEAVE (EventCallbackFunction f)
- void setCallback_IRIS_SIMULATION_TIME_EVENT (EventCallbackFunction f)
- void setConnectionInterface (IrisConnectionInterface *connection_interface)

  Set the remote connection interface.
- void **setEventHandler** (IrisInstanceEvent *handler)

  Set the event handler.
- void **setInstId** (InstanceId instId)

  Internal function. Do not call. Set the instance id of this instance. The instId is automatically set after calling instane↩ Registry_registerInstance().
- void setPendingSyncStepResponse (RequestId requestId)

  Set pending response to a step_syncStep() call.
- template<class T >
  void setProperty (const std::string &propertyName, const T &propertyValue)

  Set/add instance property.
- bool setSyncStepEventBufferId (EventBufferId evBufId)

  Set event buffer to use with step_syncStep() call.
- void setThrowOnError (bool throw_on_error)

  Set default error behavior for irisCall().
- void simulationTimeDisableEvents ()

  Disable the internal reception of IRIS_SIMULATION_TIME_EVENT events for performance reasons (e.g. during synchronous stepping).
- bool simulationTimeIsRunning ()

  Return true iff simulation is currently running.
- void simulationTimeRun ()

  Run simulation time and wait until simulation time started running.
- void simulationTimeRunUntilStop (double timeoutInSeconds=0.0)

  Run simulation time and wait until simulation time stopped again or until timeout expired.
- void simulationTimeStop ()

  Stop simulation time and wait until simulation time stopped.
- bool simulationTimeWaitForStop (double timeoutInSeconds=0.0)

  Wait for simulation time to stop or timeout.
- void unpublishCppInterface (const std::string &interfaceName)

  Unpublish a previously published C++ interface.
- void **unregisterEventBufferCallback** (const std::string &name)

  Unregister the named event buffer callback function.
- void **unregisterEventCallback** (const std::string &name)

  Unregister the named event callback function.
- void **unregisterFunction** (const std::string &name)

  Unregister a function that was previously registered with registerFunction() or irisRegisterFunction().
- IrisErrorCode unregisterInstance ()

  Unregister this instance.
- ~**IrisInstance** ()

  Destructor.

Static Public Attributes

- static const bool **ASYNCHRONOUS** = !SYNCHRONOUS

  Cause enableEvent() callback to be called back asynchronously (i.e. the caller does not wait for the function call to return).
- static const uint64_t **DEFAULT_FLAGS** = THROW_ON_ERROR

  Default flags used if not otherwise specified.
- static const bool **SYNCHRONOUS** = true

  Cause enableEvent() callback to be called back synchronously (i.e. the caller is blocked until the callback function returns).
- static const uint64_t **THROW_ON_ERROR** = (1 << 1)

  Throw an exception when an Iris call returns an error response.
- static const uint64_t **UNIQUIFY** = (1 << 0)

  Uniquify instance name when registering.

Protected Attributes

- IrisLogger **log**

  Logger.
- InstanceInfo **thisInstanceInfo** {}

  InstanceInfo of this instance.

8.17.1 Member Typedef Documentation

8.17.1.1 EventCallbackFunction

```
using iris::IrisInstance::EventCallbackFunction = std::function<IrisErrorCode(EventStreamId,
const IrisValueMap&, uint64_t, InstanceId, bool, std::string&)>
```

Event callback function type.
(Each IrisInstance can implicitly register two events which are used internally (IRIS_SIMULATION_TIME_EVENT and IRIS_SHUDOWN_LEAVE). Using the functions below clients can make use of these events without going through the effort of calling irisRegisterEventCallback()/registerEventCallback(), event_getEventSource() and eventStream_create(), and it also reduces the number of callbacks being called at runtime.

8.17.2 Constructor & Destructor Documentation

8.17.2.1 IrisInstance() [1/2]

```
iris::IrisInstance::IrisInstance (
            IrisConnectionInterface * connection_interface = nullptr,
            const std::string & instName = std::string(),
            uint64_t flags = DEFAULT_FLAGS )
```

Construct a new Iris instance.

Parameters

connection_interface	The IrisConnectionInterface that this instance should use to connect to the simulation.
instName	Name of the instance. This should be prefixed with one of the following, as appropriate: • "client." • "component." • "framework."

Parameters

flags	A bitwise OR of Instance Flags. Client instances should usually set the flag iris::IrisInstance::UNIQUIFY.

8.17.2.2 IrisInstance() [2/2]

```
iris::IrisInstance::IrisInstance (
            IrisInstantiationContext * context )
```

Construct a new Iris instance using an IrisInstantiationContext.

Parameters

context	A context object that provides the necessary information to instantiate an instance.

8.17.3 Member Function Documentation

8.17.3.1 addCallback_IRIS_INSTANCE_REGISTRY_CHANGED()

```
void iris::IrisInstance::addCallback_IRIS_INSTANCE_REGISTRY_CHANGED (
            EventCallbackFunction f )
```

Add callback function for IRIS_INSTANCE_REGISTRY_CHANGED.

8.17.3.2 destroyAllEventStreams()

```
void iris::IrisInstance::destroyAllEventStreams ( )
```

Destroy all event streams.

All event streams are always automatically destroyed when IrisInstance (and so IrisInstanceEvent) is destroyed. This function allows to destroy all event streams to be destroyed before IrisInstance.

8.17.3.3 disableEvent()

```
void iris::IrisInstance::disableEvent (
            const std::string & eventSpec )
```

Disable all matching event callback(s).

This disables all event callbacks which were previously enabled using enableEvent() which match eventSpec. The eventSpec argument for enableEvent() and disableEvent() do not have to be the same string. In particular it is not necessary to specify event fields and it is not possible to selectively disable one specific event stream out of multiple created for the same event source.

disableEvent() always iterates over all currently active event streams and disables all event streams which originate from the event sources specified in eventSpec.

Example: // Handle INST of cpu0 and cpu1 in different ways. irisInstance.enableEvent("∗.cpu0.INST", [&] (const EventStreamInfo& eventStreamInfo, IrisReceivedRequest& request) { ... }); irisInstance.enableEvent("∗.cpu1.INST", [&] (const EventStreamInfo& eventStreamInfo, IrisReceivedRequest& request) { ... }); // Disable just the cpu1 events. irisInstance.disableEvent("∗.cpu1.INST");

8.17.3.4 enableEvent() [1/2]

```
void iris::IrisInstance::enableEvent (
            const std::string & eventSpec,
            std::function< void()> callback,
            bool syncEc = false )
```

Enable event callback(s).

This is equivanet to enableEvent() specified above except that the callback does not take any arguments which is useful for the global simulation phase events.
Example:
Initialize a plugin or client in the SystemC end_of_elaboration() phase. This is the phase when all other instances are initialized and can be inspected. irisInstance.enableEvent("IRIS_SIM_PHASE_END_OF_ELABORATION", [&] { ... enable trace (using enableTrace()), inspect other instances, etc ... }, iris::IrisInstance::SYNCHRONOUS);

8.17.3.5 enableEvent() [2/2]

```
void iris::IrisInstance::enableEvent (
            const std::string & eventSpec,
            std::function< void(const EventStreamInfo &eventStreamInfo, IrisReceivedRequest
&request)> callback,
            bool syncEc = ASYNCHRONOUS )
```

Enable event callback(s).
Create one or more event streams and set up the callback function to be called for all events on the event streams. If no event stream is created because no event source matching spec is found, or if an error occurred when create an events stream, an error is thrown.
Calling this function multiple times matching the same event source is valid, but it results in multiple event streams being created which should usually be avoided for performance reasons.
A new unique callback function with the name ec_i<instanceId>_<eventSourceName>[N] is registered, where N is used to make the function name different from all other functions. This is name usually not of interest for the usage of this function.

Parameters

eventSpec	This specifies one or more event source names of one or more instances. See findEventSourcesAndFields() for the syntax specification. When the instance part of an event source is omitted the global instance is assumed. Passing "help" will throw an E_help_message error with a help messages describing the syntax and listing all available event sources in the system.
callback	Callback function called for every event. Usually a lambda function.
syncEc	If true, call callback function synchronously (i.e. caller waits for return of the callback function). Useful for simulation phases.

Examples:
Initialize a plugin or client in the SystemC end_of_elaboration() phase. This is the phase when all other instances are initialized and can be inspected. Every plugin usually does this in its constructor to enable other traces in the end↩ _of_elaboration() phase. irisInstance.enableEvent("IRIS_SIM_PHASE_END_OF_ELABORATION", [&] { // Enable traces, inspect other instances. irisInstance.enableEvent("∗.INST", [&] (const EventStreamInfo& eventStreamInfo, IrisReceivedRequest& request) { ... handle INST trace ... }); }, iris::IrisInstance::SYNCHRONOUS);
Print all simulation phases as they happen: irisInstance.enableEvent("IRIS_SIM_PHASE_∗:IRIS_SHUTDOWN_∗", [&](const iris::EventStreamInfo& eventStreamInfo, iris::IrisReceivedRequest&) { std::cout << eventStreamInfo.↩ eventSourceInfo.name << "\n"; }, iris::IrisInstance::SYNCHRONOUS);
Receive INST callbacks from all cores: irisInstance.enableEvent("∗.INST", [&] (const EventStreamInfo& event↩ StreamInfo, IrisReceivedRequest& request) { ... });
See also Examples/Plugin/SimpleTrace/main.cpp and Examples/Plugin/GenericTrace/main.cpp.
This may throw:

- E_syntax_error: Syntax error in spec (like missing closing parenthesis).
- E_unknown_event_source: A pattern in EVENT_SOURCE in eventSpec did not match any instance and/or event source name.
- E_unknown_event_field: A pattern in FIELD_OR_OPTION in eventSpec did not match any field or option of its event source.

8.17.3.6 eventBufferDestroyed()

```
void iris::IrisInstance::eventBufferDestroyed (
            EventBufferId evBufId )
```

Notify instance that a specific event buffer was just destroyed.
This function is called when a client disconnects because then all event buffers and event streams associated with that client are destroyed. It is also called by eventBuffer_destroy().
This function clears a pendingSyncStepResponse if this uses the destroyed event buffer. It also clears the evBufId cached in IrisInstanceStep if it uses the destroyed event buffer.

8.17.3.7 findEventSources()

```
std::vector< EventSourceInfo > iris::IrisInstance::findEventSources (
            const std::string & instancePathFilter = "all" )
```

Find all event sources in the system.
See filterInstanceInfos() (IrisObjects.h) for instancePathFilter semantics.

8.17.3.8 findEventSourcesAndFields()

```
std::vector< EventStreamInfo > iris::IrisInstance::findEventSourcesAndFields (
            const std::string & spec,
            InstanceId defaultInstId = IRIS_UINT64_MAX )
```

Find specific event sources in the system.
Find all event sources in the system and/or in the instance defined by defaultInstId matching wildcard patterns.
All matching event sources are added to eventStreamInfosOut which is not cleared beforehand.
The following fields in each EventStreamInfo element are set to the meta-info of the events source: sInstId, evSrcId, evSrcName, fields, hasFields and eventSourceInfo.
No event streams are created. The output is suitable as the eventStreamInfos argument for eventBuffer_create(). Alternatively, individual event streams can be created using eventStream_create() by looping over eventStream↩ InfosOut.
The set of returned event sources is defined by the filters specified in "spec" which has the following format:

- [∼]EVENT_SOURCE ["(" [FIELD_OR_OPTION ["+" FIELD_OR_OPTION] ...] ")"] [":" ...]
- EVENT_SOURCE is a wildcard pattern matching on strings of the form <instance_path>.<event_source_↩ name> (for all instances in the system) and on strings <event_source_name> for event sources of default↩ InstId.
- FIELD_OR_OPTION is either a wildcard pattern matching on field names of the selected event sources, or it is of the format OPT=VAL setting option OPT to value VAL. Use *(+OPT=VAL) to set option and still emit all fields.*
- *Use ∼EVENT_SOURCE to remove any previously matched event sources. The adding and removing event sources is executed in the specified order, so usually removes should come at the end. This makes it easy to enable events using wildcards and then exclude certain events. Example: ∗:∼∗UTLB*: Enable all events in the system except all UTLB related events.
- Likewise, use ∼FIELD to remove any previously selected fields. When the first FIELD is a negative field matching starts with all fields.

Examples:

- INST (Trace INST on the selected core.)
  " - ∗.INST:∗.CORE_STORES (Trace INST and CORE_STORES on all cores.)\n"
- ∗.INST(PC+DISASS) (Only trace PC and disassembly of INST.)
  " - ∗.INST(∼DISASS) (Trace all fields except disassembly of INST.)\n"
- ∗:∼∗SEMIHOSTING∗:∼∗UTLB∗ (Enable all trace sources in the whole system except semihosting and UTLB related traces.)
  " - ∗.TRACE_DATA_FMT_V1_1(+bufferSize=1048576) (Enable trace stream in FMT V1.1 format with buffer size 1MB and all fields.)\n\n";

This may throw:

- E_syntax_error: Syntax error in spec (like missing closing parenthesis).
- E_unknown_event_source: A pattern in EVENT_SOURCE in spec did not match any instance and/or event source name.
- E_unknown_event_field: A pattern in FIELD_OR_OPTION in spec did not match any field or option of its event source.

8.17.3.9 findInstanceInfos()

```
std::vector< InstanceInfo > iris::IrisInstance::findInstanceInfos (
            const std::string & instancePathFilter = "all" )
```

Find instance infos of all instances in the system.
This function uses instance info data cached in this instance. The cache can be cleared with clearCachedMetaInfo().
See filterInstanceInfos() (IrisObjects.h) for instancePathFilter semantics.

8.17.3.10 getBuilder()

```
IrisInstanceBuilder * iris::IrisInstance::getBuilder ( )
```

Get the IrisInstanceBuilder object for this instance. This can be used to set up metadata and callbacks for standard Iris functions.

Returns

The IrisInstanceBuilder object for this instance.

8.17.3.11 getInstanceId()

```
InstanceId iris::IrisInstance::getInstanceId (
            const std::string & instName )
```

Get instance id for a specifid instance name.
If no such instance is known IrisErrorException(E_unknown_instance_name) is thrown.
This information is cached in this instance. The cache can be cleared with clearCachedMetaInfo().

Returns

Instance id.

8.17.3.12 getInstanceInfo() [1/2]

```
InstanceInfo iris::IrisInstance::getInstanceInfo (
            const std::string & instancePathFilter )
```

Get instance info of a specific instance in the system.
This function expects either a correct instance path or a pattern which just matches a single instance, for example "core" which always returns the first core, regardless of the number of cores in the system. If no instance is found or if more than one instances are found, IrisErrorException(E_unknown_instance_name) is thrown.
This function should only be used when the instance name is known upfront, or to get access to the first core only. Use findInstanceInfos() to discover arbitrary instances.
This function uses instance info data cached in this instance. The cache can be cleared with clearCachedMetaInfo().
See filterInstanceInfos() (IrisObjects.h) for instancePathFilter semantics.

8.17.3.13 getInstanceInfo() [2/2]

```
const InstanceInfo & iris::IrisInstance::getInstanceInfo (
            InstanceId instId )
```

Get InstanceInfo including properties for a specific instId.
This information is cached in this instance. The cache can be cleared with clearCachedMetaInfo().

Returns

InstanceInfo (including properties) for instId. Throws IrisErrorException(E_unknown_instance_id) if instId is unknown.

8.17.3.14 getInstanceList()

```
const std::vector< InstanceInfo > & iris::IrisInstance::getInstanceList ( )
```

Get list of InstanceInfos of all instances in the system, including properties.
Note that the index into the returned list is generally not the InstanceId. Use getInstanceInfo(instId) to get the InstanceInfo for a specific instance id.
This information is cached in this instance. The cache can be cleared with clearCachedMetaInfo().

Returns

InstanceInfos (including properties) for all instances in the system.

8.17.3.15 getInstanceName() [1/2]

```
const std::string & iris::IrisInstance::getInstanceName ( ) const  [inline]
```

Get the instance name of this instance. This is valid after registerInstance() returns.

Returns

The instance name of this instance. This is the same as the name parameter passed to the constructor or registerInstance() unless this instance was registered with the UNIQUIFY flag set and the name was modified to make it unique.

8.17.3.16 getInstanceName() [2/2]

```
std::string iris::IrisInstance::getInstanceName (
            InstanceId instId )
```

Get instance name for a specifid instId.
This function does not throw. It returns "instance.<instId>" for unknown instIds.
This information is cached in this instance. The cache can be cleared with clearCachedMetaInfo().

Returns

instance name or "instance.<instId>" instId is unknown.

8.17.3.17 getInstId()

```
InstanceId iris::IrisInstance::getInstId ( ) const  [inline]
```

Get the instance id of this instance. This is valid after registerInstance() returns.

Returns

The instId for this instance.

8.17.3.18 getLocalIrisInterface()

```
IrisInterface * iris::IrisInstance::getLocalIrisInterface ( )  [inline]
```

Get the local IrisInterface of this instance. This is the interface that other instances use to send their requests and responses to this instance.

Returns

IrisInterface to send messages to this instance.

8.17.3.19 getLogger()

```
IrisLogger & iris::IrisInstance::getLogger ( )  [inline]
```

Get logger.
This can be used by add-ons and owners to log on behalf (using the name of) this instance.

8.17.3.20 getMemorySpaceId()

```
MemorySpaceId iris::IrisInstance::getMemorySpaceId (
            InstanceId instId,
            const std::string & name )
```

Get memory space id of memory space by name.
Note: Memory space names change over time and are not a stable method to identify memory spaces. If possible the canonical memory space number should be used instead to identify memory spaces.
See getMemorySpaceInfo() for all supported name formats.

8.17.3.21 getMemorySpaceInfo()

```
const MemorySpaceInfo & iris::IrisInstance::getMemorySpaceInfo (
            InstanceId instId,
            const std::string & name )
```

Get MemorySpaceInfo of memory space by name.
Note: Memory space names change over time and are not a stable method to identify memory spaces. If possible the canonical memory space number should be used instead to identify memory spaces.
Supported formats for name:

- <name> : Memory space identified by its name.
- cmn:<CanonicalMemorySpaceNumber> : Specify memory space by it canonical memory space number, e.g. cmn:0x10ff for "current" virtual memory space.
- id:<MemorySpaceId> : Specify memory space by its memory space id.

8.17.3.22 getPropertyMap()

```
const PropertyMap & iris::IrisInstance::getPropertyMap ( ) const  [inline]
```

Get property map.
This can be used to lookup properties: getWithDefault(my_instance->getPropertyMap(), "myStringProperty", "").getAsString();

8.17.3.23 getRemoteIrisInterface()

```
IrisInterface * iris::IrisInstance::getRemoteIrisInterface ( )  [inline]
```

Get the remote Iris interface.

Returns

Returns the IrisInterface that this instance sends requests and responses to.

8.17.3.24 getResourceId()

```
ResourceId iris::IrisInstance::getResourceId (
            InstanceId instId,
            const std::string & resourceSpec )
```

Get resource id for a specific resource.
See resourceRead() for semantics of resourceSpec.
Throws an error when resource is not found.

Returns

Resource id.

8.17.3.25 getResourceInfo()

```
const ResourceInfo & iris::IrisInstance::getResourceInfo (
            InstanceId instId,
            const std::string & resourceSpec )
```

Get ResourceInfo for a specific resource.
See resourceRead() for semantics of resourceSpec, errors and limitations.

8.17.3.26 getResourceInfos()

```
std::vector< ResourceInfo > iris::IrisInstance::getResourceInfos (
            InstanceId instId,
            const std::string & resourceSpec )
```

Get zero or more matching resource infos of an instance.
resourceSpec may contain wildcards. To get all resource infos use getResourceInfos(InstanceId instId).
See resourceRead() for semantics of resourceSpec, errors and limitations.

8.17.3.27 irisCall()

```
IrisCppAdapter & iris::IrisInstance::irisCall ( )  [inline]
```

Get an IrisCppAdapter to call an Iris function of any other instance.
Usage:

```
irisCall().resource_read(...);
```

for the Iris function `resource_read()`.

8.17.3.28 irisCallNoThrow()

```
IrisCppAdapter & iris::IrisInstance::irisCallNoThrow ( )  [inline]
```

Get an IrisCppAdapter to call an Iris function of any other instance.
When an Iris function returns an error response, this adapter returns the error code and does not throw an exception.
Usage:

```
iris::IrisErrorCode code = irisCallNoThrow().resource_read(...);
```

8.17.3.29 irisCallThrow()

```
IrisCppAdapter & iris::IrisInstance::irisCallThrow ( )  [inline]
```

Get an IrisCppAdapter to call an Iris function of any other instance. When an Iris function returns an error response, this adapter always throws an exception. Usage:

```
try
{
    irisCall().resource_read(...);
}
catch (iris::IrisErrorException &e)
{
    ...
}
```

8.17.3.30 isEventEnabled()

```
bool iris::IrisInstance::isEventEnabled (
            const std::string & eventSpec )
```

Check whether a certain event is already enabled through enableEvent.
This is useful for code which wants to initialize event handling on demand, so that this code does not need to maintains its own "did I already initialize myself" variable.

8.17.3.31 isRegistered()

```
bool iris::IrisInstance::isRegistered ( ) const  [inline]
```

Return true iff we are registered as an instance (= we have a valid instance id).

8.17.3.32 isValidEvBufId()

```
bool iris::IrisInstance::isValidEvBufId (
            EventBufferId evBufId ) const
```

Check whether event buffer id is valid.
This function is use to validate event buffer ids.

Returns

Returns true iff evBufId is a valid event buffer id.

8.17.3.33 notifyStateChanged()

```
void iris::IrisInstance::notifyStateChanged ( )
```

Notify client instances that the state of any resource/memory/table/disassembly etc changed.
This should only ever be called when the value of anything changes spontaneously, e.g. through a private GUI of an instance. This must not be called when the state changes because of normal simulation operations.
Calling this function is very exotic. Normal component instances and client instances will never want to call this.

8.17.3.34 publishCppInterface()

```
template<class T >
void iris::IrisInstance::publishCppInterface (
            const std::string & interfaceName,
            T * pointer,
            const std::string & jsonDescription )  [inline]
```

Publish a C++ interface XYZ through a new instance_getCppInterfaceXYZ() function.
Null pointers are silently ignored. An interface previously registered under the same name is silently overwritten.

Parameters

interfaceName	Class name or interface name of the interface to be published. This must be a C identifier without namespaces etc. The interface can betreieved with "instance_getCppInterface<interfaceName>()".
pointer	Pointer to the C++ class instance implementing this interface.
jsonDescription	Text for FunctionInfo.description. This must be a valid JSON string without enclosing quotes. This text is amended by generic notes aboud the compatibility of C++ pointers which are valid for every C++ interface.

8.17.3.35 registerEventBufferCallback() [1/3]

```
void iris::IrisInstance::registerEventBufferCallback (
            EventBufferCallbackDelegate delegate,
```

```
            const std::string & name,
            const std::string & description,
            const std::string & dlgInstanceTypeStr )  [inline]
```

Register an event buffer callback using an EventBufferCallbackDelegate.

Parameters

delegate	EventBufferCallbackDelegate to call to handle the function.
name	Name of the function as it will be published.
description	Description of this event callback function.
dlgInstanceTypeStr	The name of the delegate type. This is only used for logging purposes.

8.17.3.36 registerEventBufferCallback() [2/3]

```
template<typename T , IrisErrorCode(T::*)(const EventBufferCallbackData &data) METHOD>
void iris::IrisInstance::registerEventBufferCallback (
            T * instance,
            const std::string & name,
            const std::string & description,
            const std::string & dlgInstanceTypeStr )  [inline]
```

Register an event buffer callback using an EventBufferCallbackDelegate.

Parameters

instance	An instance of class T on which to call the delegate T::METHOD().
name	Name of the function as it will be published.
description	Description of this event callback function.
dlgInstanceTypeStr	The name of the delegate type. This is only used for logging purposes.

8.17.3.37 registerEventBufferCallback() [3/3]

```
template<class T >
void iris::IrisInstance::registerEventBufferCallback (
            T * instance,
            const std::string & name,
            const std::string & description,
            void(T::*)(IrisReceivedRequest &) memberFunctionPtr,
            const std::string & instanceTypeStr )  [inline]
```

Register an event buffer callback function.
Event buffer callbacks have the same signature, only the description is different.

Parameters

instance	An instance of class T on which to call the member function.
name	Name of the function as it will be published.
description	Description of this event callback function.
memberFunctionPtr	Pointer to the C++ implementation of the function.
instanceTypeStr	The name of class T. This is only used for logging purposes.

8.17.3.38 registerEventCallback() [1/3]

```
void iris::IrisInstance::registerEventCallback (
            EventCallbackDelegate delegate,
            const std::string & name,
            const std::string & description,
            const std::string & dlgInstanceTypeStr )  [inline]
```

Register a general event callback using an EventCallbackDelegate.

Parameters

delegate	EventCallbackDelegate to call to handle the function.
name	Name of the function as it will be published.
description	Description of this event callback function.
dlgInstanceTypeStr	The name of the delegate type. This is only used for logging purposes.

8.17.3.39 registerEventCallback() [2/3]

```
template<typename T , IrisErrorCode(T::*)(uint64_t, const AttributeValueMap &, uint64_t, uint64↩
_t, bool, std::string &) METHOD>
void iris::IrisInstance::registerEventCallback (
            T * instance,
            const std::string & name,
            const std::string & description,
            const std::string & dlgInstanceTypeStr )  [inline]
```

Register a general event callback using an EventCallbackDelegate.

Parameters

instance	An instance of class T on which to call the delegate T::METHOD().
name	Name of the function as it will be published.
description	Description of this event callback function.
dlgInstanceTypeStr	The name of the delegate type. This is only used for logging purposes.

8.17.3.40 registerEventCallback() [3/3]

```
template<class T >
void iris::IrisInstance::registerEventCallback (
            T * instance,
            const std::string & name,
            const std::string & description,
            void(T::*)(IrisReceivedRequest &) memberFunctionPtr,
            const std::string & instanceTypeStr )  [inline]
```

Register a general event callback.
Event callbacks have the same signature, only the description is different.

Parameters

instance	An instance of class T on which to call the member function.
name	Name of the function as it will be published.
description	Description of this event callback function.
memberFunctionPtr	Pointer to the C++ implementation of the function.
instanceTypeStr	The name of class T. This is only used for logging purposes.

8.17.3.41 registerFunction()

```
template<class T >
void iris::IrisInstance::registerFunction (
            T * instance,
            const std::string & name,
            void(T::*)(IrisReceivedRequest &) memberFunctionPtr,
            const std::string & functionInfoJson,
            const std::string & instanceTypeStr )  [inline]
```

Register an Iris function implementation.
The following macro can be used instead of calling this function to avoid specifying the function name twice↩ : irisRegisterFunction(instancePtr, instanceType, functionName, functionInfoJson)

Parameters

instance	An instance of class T on which to call the member function.
name	Name of the function as it will be published.
memberFunctionPtr	Pointer to the C++ implementation of the function.
functionInfoJson	A string containing the JSON-encoded FunctionInfo object for this function.
instanceTypeStr	The name of class T. This is only used for logging purposes.

8.17.3.42 registerInstance()

```
IrisErrorCode iris::IrisInstance::registerInstance (
            const std::string & instName,
            uint64_t flags = DEFAULT_FLAGS )
```

Register this instance if it was not registered when constructed.

Parameters

instName	Name of the instance. This should be prefixed with one of the following, as appropriate: • "client." • "component." • "framework."
flags	A bitwise OR of Instance Flags. Client instances should usually set the flag iris::IrisInstance::UNIQUIFY.

8.17.3.43 resourceRead()

```
uint64_t iris::IrisInstance::resourceRead (
            InstanceId instId,
            const std::string & resourceSpec )
```

Read numeric resource and return its value.
Resource spec may be:

- <resource_name>[.<child_name>...]
- <resource_group>.<resource_name>[.<child_name>...]
- tag:<tag> (e.g. "tag:isInstructionCounter" or "tag:isPc")

- crn:<canonical_register_number_in_decimal> (usage: resourceRead(instId, "crn:" + std::to_string(iris::ElfDwarf::ARM_R0)), see iris/IrisElfDwarfArm.h, consider using resourceReadCrn() instead)
- rscId:<resourceId> (fallback in case resourceId is already known, consider using irisCallThrow()->resource_read() instead)
- resourceSpec may contain wildcards, but all functions except getResourceInfos() require the pattern to match exactly one resource.

If the resoure is not found or could not be read the appropriate error is thrown. If the resource is not a numeric resource E_type_mismatch is thrown.
This is a convenience function, intended to make reading individual well-known registers easy (e.g. PC, instruction counter). Each call issues an individual resource_read() call. When reading/writing more than one resource use IrisResourceSet or resource_read() instead since each resource_read() call can read many resources at once which is faster.
The resource meta-information is cached in this instance, but the value is not. The cache can be cleared with clearCachedMetaInfo().

Returns

Resource value.

8.17.3.44 resourceReadCrn()

```
uint64_t iris::IrisInstance::resourceReadCrn (
            InstanceId instId,
            uint64_t canonicalRegisterNumber )  [inline]
```

Read numeric resource and return its value (using the canonical register number aka DWARF register id).
See resourceRead() and the "crn:" case within.

Returns

Resource value.

8.17.3.45 resourceReadStr()

```
std::string iris::IrisInstance::resourceReadStr (
            InstanceId instId,
            const std::string & resourceSpec )
```

Read string resource, or read other resources as string.
Numeric resource values get converted to a string according to the type and bitWidth. Errors in the result.error fields are returned as string. noValue resources return "(noValue)".
See resourceRead() for semantics of resourceSpec, errors and limitations.

8.17.3.46 resourceReadWide()

```
std::vector< uint64_t > iris::IrisInstance::resourceReadWide (
            InstanceId instId,
            const std::string & resourceSpec )
```

Read wide numeric resource and return its value.
See resourceRead() for info on resourceSpec.

Returns

Resource value.

8.17.3.47 resourceWrite() [1/2]

```
void iris::IrisInstance::resourceWrite (
            InstanceId instId,
            const std::string & resourceSpec,
            const std::vector< uint64_t > & value )
```

Write wide numeric resource.
If the resource is not a numeric resource E_type_mismatch is thrown.
See resourceRead() for semantics of resourceSpec, errors and limitations.

8.17.3.48 resourceWrite() [2/2]

```
void iris::IrisInstance::resourceWrite (
            InstanceId instId,
            const std::string & resourceSpec,
            uint64_t value )
```

Write numeric resource.
If the resource is not a numeric resource E_type_mismatch is thrown.
See resourceRead() for semantics of resourceSpec, errors and limitations.

8.17.3.49 resourceWriteCrn()

```
void iris::IrisInstance::resourceWriteCrn (
            InstanceId instId,
            uint64_t canonicalRegisterNumber,
            uint64_t value )  [inline]
```

Write numeric resource by canonical register number (aka DWARF register id).
See resourceWrite() for semantics.

8.17.3.50 resourceWriteStr()

```
void iris::IrisInstance::resourceWriteStr (
            InstanceId instId,
            const std::string & resourceSpec,
            const std::string & value )
```

Write string resource, or write numeric resource from string.
If the resource is not a string the value is converted to a numeric value according to the resource type.
See resourceRead() for semantics of resourceSpec, errors and limitations.

8.17.3.51 sendRequest()

```
bool iris::IrisInstance::sendRequest (
            IrisRequest & req )  [inline]
```

Send an Iris request or notification and potentially wait for a response.

Parameters

req	Iris request to send.

Returns

Returns true iff a non-error response was received, and therefore the result values must be decoded.

Use this to manually call functions implemented in the called target but not implemented in IrisCppAdapter.

8.17.3.52 sendResponse()

```
void iris::IrisInstance::sendResponse (
            const uint64_t * response )  [inline]
```

Send a response to the remote Iris interface.

Call this from the function implementations registered with registerFunction() or irisRegisterFunction().

Parameters

response	The Iris response message to send.

8.17.3.53 setCallback_IRIS_SHUTDOWN_LEAVE()

```
void iris::IrisInstance::setCallback_IRIS_SHUTDOWN_LEAVE (
            EventCallbackFunction f )
```

Set callback function for IRIS_SHUTDOWN_LEAVE.

8.17.3.54 setCallback_IRIS_SIMULATION_TIME_EVENT()

```
void iris::IrisInstance::setCallback_IRIS_SIMULATION_TIME_EVENT (
            EventCallbackFunction f )
```

Set callback function for IRIS_SIMULATION_TIME_EVENT.

8.17.3.55 setConnectionInterface()

```
void iris::IrisInstance::setConnectionInterface (
            IrisConnectionInterface * connection_interface )
```

Set the remote connection interface.
Used to set the IrisConnectionInterface if it was not set in the constructor.

Parameters

connection_interface	The interface used to connect to an Iris simulation.

8.17.3.56 setPendingSyncStepResponse()

```
void iris::IrisInstance::setPendingSyncStepResponse (
            RequestId requestId )
```

Set pending response to a step_syncStep() call.
This function is called when the step_syncStep() function is called and the response is delivered when the simulation time stopped.

8.17.3.57 setProperty()

```
template<class T >
void iris::IrisInstance::setProperty (
            const std::string & propertyName,
            const T & propertyValue )  [inline]
```

Set/add instance property.
This creates a new property or overwrites an existing one.
Properties (name and value) are defined by the instance that has them. Properties are not to be confused with parameters, whose values are defined by clients or by parent components and some parameters might change at runtime.
Properties are exposed by the function instance_getProperties(). This should only ever be called upon initialization, before other components have a chance to call instance_getProperties(). Properties are constant and should not be changed at runtime. T can be bool, uint64_t, int64_t, or std::string.

Parameters

propertyName	Name of the property.

Parameters

propertyValue	Value of the property.

8.17.3.58 setSyncStepEventBufferId()

```
bool iris::IrisInstance::setSyncStepEventBufferId (
            EventBufferId evBufId )
```

Set event buffer to use with step_syncStep() call.
Specifying IRIS_UINT64_MAX is valid and means that step_syncStep() should not return any events".

8.17.3.59 setThrowOnError()

```
void iris::IrisInstance::setThrowOnError (
            bool throw_on_error )  [inline]
```

Set default error behavior for irisCall().

Parameters

throw_on_error	If true, calls made using irisCall() that respond with an error response will throw an exception. This is the same behavior as irisCallThrow(). If false, calls made using irisCall() that respond with an error response will return the error code and not throw an exception. This is the same behavior as irisCallNoThrow().

8.17.3.60 simulationTimeDisableEvents()

```
void iris::IrisInstance::simulationTimeDisableEvents ( )
```

Disable the internal reception of IRIS_SIMULATION_TIME_EVENT events for performance reasons (e.g. during synchronous stepping).
The callback set with setCallback_IRIS_SIMULATION_TIME_EVENT() will no longer be called.
Internal IRIS_SIMULATION_TIME_EVENTs will automatically be re-enabled as soon as one of the other simulationTime∗() functions is called.
This function throws Iris errors.

8.17.3.61 simulationTimeIsRunning()

```
bool iris::IrisInstance::simulationTimeIsRunning ( )
```

Return true iff simulation is currently running.
Note that this information is always out of date if there is another simulation controller.
This function throws Iris errors.

8.17.3.62 simulationTimeRun()

```
void iris::IrisInstance::simulationTimeRun ( )
```

Run simulation time and wait until simulation time started running.
Does not wait until model stopped again. See simulationTimeRunUntilStop().
This function throws Iris errors.

8.17.3.63 simulationTimeRunUntilStop()

```
void iris::IrisInstance::simulationTimeRunUntilStop (
            double timeoutInSeconds = 0.0 )
```

Run simulation time and wait until simulation time stopped again or until timeout expired.
This function throws Iris errors.

8.17.3.64 simulationTimeStop()

```
void iris::IrisInstance::simulationTimeStop ( )
```

Stop simulation time and wait until simulation time stopped.
This function throws Iris errors.

8.17.3.65 simulationTimeWaitForStop()

```
bool iris::IrisInstance::simulationTimeWaitForStop (
            double timeoutInSeconds = 0.0 )
```

Wait for simulation time to stop or timeout.
This function only works after simulationTimeRun() has been called. When the simulation time already stopped after simulationTimeRun() then this function exits immediately.
This function throws Iris errors.

Parameters

timeoutInSeconds	Stop waiting after the specified timeout and return false on timeout. 0.0 means to wait indefinitely.

Returns

true if simulation time stopped, false on timeout. When timeoutInSeconds is 0.0 (= no timeout) this always returns true.

8.17.3.66 unpublishCppInterface()

```
void iris::IrisInstance::unpublishCppInterface (
            const std::string & interfaceName )  [inline]
```

Unpublish a previously published C++ interface.
After calling this function the corresponding instance_getCppInterface...() function is no longer available. This is silently ignored If the interface was not previously published.

Parameters

interfaceName	Class name or interface name of the interface to be unpublished.

8.17.3.67 unregisterInstance()

```
IrisErrorCode iris::IrisInstance::unregisterInstance ( )
```

Unregister this instance.
Iris calls must not be made after the instance has been unregistered.
The documentation for this class was generated from the following file:

- IrisInstance.h

8.18 iris::IrisInstanceBreakpoint Class Reference

Breakpoint add-on for IrisInstance.
`#include <IrisInstanceBreakpoint.h>`

Public Member Functions

- void addCondition (const std::string &name, const std::string &type, const std::string &description, const std::vector< std::string > bpt_types=std::vector< std::string >())

Add an optional component-specific condition that can be configured by clients.

- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- const BreakpointInfo * getBreakpointInfo (BreakpointId bptId) const

  Get BreakpointInfo for a breakpoint id.
- void handleBreakpointHits (const BreakpointHitInfos &hitBpts)

  Handle breakpoint hit.
- bool **hasAnyBreakpointSet** ()
- **IrisInstanceBreakpoint** (IrisInstance *irisInstance=nullptr)
- void notifyBreakpointHit (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId)

  Notify clients that a code breakpoint was hit.
- void notifyBreakpointHitData (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId, uint64_t accessAddr, uint64_t accessSize, const std::string &accessRw, const std::vector< uint64_t > &data)

  Notify clients that a data breakpoint was hit.
- void notifyBreakpointHitRegister (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId, const std::string &accessRw, const std::vector< uint64_t > &data)

  Notify clients that a register breakpoint was hit.
- void setBreakpointDeleteDelegate (BreakpointDeleteDelegate delegate)

  Set breakpoint delete delegate for all breakpoints deleted by this instance.
- void setBreakpointSetDelegate (BreakpointSetDelegate delegate)

  Set breakpoint set delegate for all breakpoints set by this instance.
- void setEventHandler (IrisInstanceEvent *handler)

  Set the event handler used to notify the clients that enable the IRIS_BREAKPOINT_HIT event.
- void setHandleBreakpointHitsDelegate (std::function< IrisErrorCode(const BreakpointHitInfos &hitBpts)> delegate)

  Set a delegate for handling breakpoint hit in this instance.

8.18.1 Detailed Description

Breakpoint add-on for IrisInstance.
Instances use this class to support breakpoint functionality.
It implements all Iris breakpoint*() functions and maintains the breakpoint information that is set by breakpoint_set() and is exposed by breakpoint_getList().
Example usage:

```
irisInstanceBpt = new iris::IrisInstanceBreakpoint(irisInstance);
irisInstanceBpt->setBreakpointSetDelegate(bptSetDel);           // Use this delegate for breakpoint set.
irisInstanceBpt->setBreakpointDeleteDelegate(bptDeleteDel);     // Use this delegate for breakpoint delete.
// When a breakpoint is hit, notify the instances that enable the IRIS_BREAKPOINT_HIT event.
irisInstanceBpt->setEventHandler(irisInstanceEvent);
```

See DummyComponent.h for a working example.

8.18.2 Member Function Documentation

8.18.2.1 addCondition()

```
void iris::IrisInstanceBreakpoint::addCondition (
            const std::string & name,
            const std::string & type,
            const std::string & description,
            const std::vector< std::string > bpt_types = std::vector< std::string >() )
```

Add an optional component-specific condition that can be configured by clients.

Parameters

name	The name of the condition.

Parameters

type	The type of the value that clients set to configure the condition.
description	A description of the condition.
bpt_types	A list of breakpoint types that this condition can be applied to. An empty list indicates all types.

8.18.2.2 attachTo()

```
void iris::IrisInstanceBreakpoint::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.
Only use this method if nullptr was passed to the constructor.

Parameters

irisInstance	The IrisInstance to attach to.

8.18.2.3 getBreakpointInfo()

```
const BreakpointInfo * iris::IrisInstanceBreakpoint::getBreakpointInfo (
            BreakpointId bptId ) const
```

Get BreakpointInfo for a breakpoint id.

Parameters

bpt↩ Id	The breakpoint id for which the BreakpointInfo is requested.

Returns

A pointer to the BreakpointInfo for the requested breakpoint or nullptr if *bptId* is not a valid breakpoint id.

8.18.2.4 handleBreakpointHits()

```
void iris::IrisInstanceBreakpoint::handleBreakpointHits (
            const BreakpointHitInfos & hitBpts )
```

Handle breakpoint hit.

Parameters

hitBpts	The information of the breakpoint that is hit. Calls a delegate method in the model.

8.18.2.5 notifyBreakpointHit()

```
void iris::IrisInstanceBreakpoint::notifyBreakpointHit (
            BreakpointId bptId,
            uint64_t time,
            uint64_t pc,
            MemorySpaceId pcSpaceId )
```

Notify clients that a code breakpoint was hit.

It notifies clients by emitting an IRIS_BREAKPOINT_HIT event.

Parameters

bptId	Breakpoint id for the breakpoint that was hit.
time	Simulation time at which the breakpoint hit.
pc	Value of the relevant program counter when the event hit.
pc↩ SpaceId	Memory space Id for the memory space that the PC address corresponds to.

8.18.2.6 notifyBreakpointHitData()

```
void iris::IrisInstanceBreakpoint::notifyBreakpointHitData (
            BreakpointId bptId,
            uint64_t time,
            uint64_t pc,
            MemorySpaceId pcSpaceId,
            uint64_t accessAddr,
            uint64_t accessSize,
            const std::string & accessRw,
            const std::vector< uint64_t > & data )
```

Notify clients that a data breakpoint was hit.
It notifies clients by emitting an IRIS_BREAKPOINT_HIT event.

Parameters

bptId	Breakpoint id for the breakpoint that was hit.
time	Simulation time at which the breakpoint hit.
pc	Value of the relevant program counter when the event hit.
pcSpaceId	Memory space Id for the memory space that the PC address corresponds to.
accessAddr	The address of the data access that triggered the breakpoint.
accessSize	The size of the data access that triggered the breakpoint.
accessRw	Indicates the direction of the access. "r" = read access or "w" = write access.
data	The data that was written or read during the access that triggered the breakpoint.

8.18.2.7 notifyBreakpointHitRegister()

```
void iris::IrisInstanceBreakpoint::notifyBreakpointHitRegister (
            BreakpointId bptId,
            uint64_t time,
            uint64_t pc,
            MemorySpaceId pcSpaceId,
            const std::string & accessRw,
            const std::vector< uint64_t > & data )
```

Notify clients that a register breakpoint was hit.
It notifies clients by emitting an IRIS_BREAKPOINT_HIT event.

Parameters

bptId	Breakpoint id for the breakpoint that was hit.
time	Simulation time at which the breakpoint hit.
pc	Value of the relevant program counter when the event hit.

Parameters

pc↩ SpaceId	Memory space Id for the memory space that the PC address corresponds to.
accessRw	Indicates the direction of the access. "r" = read access or "w" = write access.
data	The data that was written or read during the access that triggered the breakpoint.

8.18.2.8 setBreakpointDeleteDelegate()

```
void iris::IrisInstanceBreakpoint::setBreakpointDeleteDelegate (
            BreakpointDeleteDelegate delegate )
```

Set breakpoint delete delegate for all breakpoints deleted by this instance.

Parameters

delegate	A BreakpointDeleteDelegate to call when a breakpoint is deleted.

8.18.2.9 setBreakpointSetDelegate()

```
void iris::IrisInstanceBreakpoint::setBreakpointSetDelegate (
            BreakpointSetDelegate delegate )
```

Set breakpoint set delegate for all breakpoints set by this instance.

Parameters

delegate	A BreakpointSetDelegate to call when a breakpoint is set.

8.18.2.10 setEventHandler()

```
void iris::IrisInstanceBreakpoint::setEventHandler (
            IrisInstanceEvent * handler )
```

Set the event handler used to notify the clients that enable the IRIS_BREAKPOINT_HIT event.
All breakpoint events are normal events and are handled through the same mechanism as other events.

8.18.2.11 setHandleBreakpointHitsDelegate()

```
void iris::IrisInstanceBreakpoint::setHandleBreakpointHitsDelegate (
            std::function< IrisErrorCode(const BreakpointHitInfos &hitBpts)> delegate )
```

Set a delegate for handling breakpoint hit in this instance.

Parameters

delegate	A function to call when breakpoint(s) are hit.

The documentation for this class was generated from the following file:

- IrisInstanceBreakpoint.h

8.19 iris::IrisInstanceBuilder Class Reference

Builder interface to populate an IrisInstance with registers, memory etc.

```
#include <IrisInstanceBuilder.h>
```

Classes

- class AddressTranslationBuilder

  Used to set metadata for an address translation.
- class EventSourceBuilder

  Used to set metadata on an EventSource.
- class FieldBuilder

  Used to set metadata on a register field resource.
- class MemorySpaceBuilder

  Used to set metadata for a memory space.
- class ParameterBuilder

  Used to set metadata on a parameter.
- class RegisterBuilder

  Used to set metadata on a register resource.
- struct RegisterEventEmitterPair
- class SemihostingManager

  semihosting_apis IrisInstanceBuilder semihosting APIs
- class TableBuilder

  Used to set metadata for a table.
- class TableColumnBuilder

  Used to set metadata for a table column.

Public Types

- typedef std::map< uint64_t, RegisterEventEmitterPair > **RscIdEventEmitterMap**

Public Member Functions

- AddressTranslationBuilder addAddressTranslation (MemorySpaceId inSpaceId, MemorySpaceId outSpace↩ Id, const std::string &description)

  Add an address translation.
- void **addBreakpointCondition** (const std::string &name, const std::string &type, const std::string &description, const std::vector< std::string > bpt_types=std::vector< std::string >())

  Add an optional component-specific condition.
- EventSourceBuilder addEventSource (const std::string &name, bool isHidden=false)

  Add metadata for an event source.
- EventSourceBuilder addEventSource (const std::string &name, IrisEventEmitterBase &event_emitter, bool isHidden=false)

  Add metadata for an event source that uses an IrisEventEmitter.
- MemorySpaceBuilder addMemorySpace (const std::string &name)

  Add metadata for one memory space.
- RegisterBuilder addNoValueRegister (const std::string &name, const std::string &description, const std::string &format)

  Add metadata for one noValue resource.
- ParameterBuilder addParameter (const std::string &name, uint64_t bitWidth, const std::string &description)

  Add numeric parameter.
- RegisterBuilder addRegister (const std::string &name, uint64_t bitWidth, const std::string &description, uint64_t addressOffset=IRIS_UINT64_MAX, uint64_t canonicalRn=IRIS_UINT64_MAX)

  Add metadata for one numeric register resource.
- ParameterBuilder addStringParameter (const std::string &name, const std::string &description)

  Add string parameter.

- RegisterBuilder addStringRegister (const std::string &name, const std::string &description)

  Add metadata for one string register resource.
- TableBuilder addTable (const std::string &name)

  Add metadata for one table.
- void beginResourceGroup (const std::string &name, const std::string &description, uint64_t subRscIdStart=IRIS_UINT64_MAX, const std::string &cname=std::string())

  Begin a new resource group.
- void deleteEventSource (const std::string &name)

  Delete event source.
- EventSourceBuilder enhanceEventSource (const std::string &name)

  Enhance existing event source.
- ParameterBuilder enhanceParameter (ResourceId rscId)

  Get ParameterBuilder to enhance a parameter.
- RegisterBuilder enhanceRegister (ResourceId rscId)

  Get RegisterBuilder to enhance register.
- void finalizeRegisterReadEvent ()
- void finalizeRegisterUpdateEvent ()

  Finalize set up of an IrisEventEmitter.
- const BreakpointInfo * getBreakpointInfo (BreakpointId bptId)

  Get the breakpoint information for a given breakpoint.
- IrisInstanceEvent * getIrisInstanceEvent ()
- RscIdEventEmitterMap getRegisterEventEmitterMap ()

  Returns event emitters associated with registers.
- const ResourceInfo & getResourceInfo (ResourceId rscId)

  Get ResourceInfo of a previously added register.
- bool hasEventSource (const std::string &name)

  Check whether event source already exists.
- IrisInstanceBuilder (IrisInstance *iris_instance)

  Construct an IrisInstanceBuilder for an Iris instance.
- void notifyBreakpointHit (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId)

  Notify clients that a code breakpoint was hit.
- void notifyBreakpointHitData (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId, uint64_t accessAddr, uint64_t accessSize, const std::string &accessRw, const std::vector< uint64_t > &data)

  Notify clients that a data breakpoint was hit (`IRIS_BREAKPOINT_HIT`).
- void notifyBreakpointHitRegister (BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId, const std::string &accessRw, const std::vector< uint64_t > &data)

  Notify clients that a register breakpoint was hit (`IRIS_BREAKPOINT_HIT`).
- uint64_t openImage (const std::string &filename)

  Open an image to be read using image_loadDataPull() or image_loadDataRead().
- void renameEventSource (const std::string &name, const std::string &newName)

  Rename existing event source.
- void resetRegisterReadEvent ()

  Reset the active register read event.
- void resetRegisterUpdateEvent ()

  Reset the active register update event.
- template<IrisErrorCode(*)(const BreakpointInfo &) FUNC>
  void setBreakpointDeleteDelegate ()

  Set the delegate that is called when a breakpoint is deleted.
- void setBreakpointDeleteDelegate (BreakpointDeleteDelegate delegate)

  Set the delegate that is called when a breakpoint is deleted.

- template<typename T , IrisErrorCode(T::*)(const BreakpointInfo &) METHOD>
  void setBreakpointDeleteDelegate (T *instance)

  Set the delegate that is called when a breakpoint is deleted.
- template<IrisErrorCode(*)(BreakpointInfo &) FUNC>
  void setBreakpointSetDelegate ()

  Set the delegate that is called when a breakpoint is set.
- void setBreakpointSetDelegate (BreakpointSetDelegate delegate)

  Set the delegate that is called when a breakpoint is set.
- template<typename T , IrisErrorCode(T::*)(BreakpointInfo &) METHOD>
  void setBreakpointSetDelegate (T *instance)

  Set the delegate that is called when a breakpoint is set.
- template<IrisErrorCode(*)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &) FUNC>
  void setDefaultAddressTranslateDelegate ()

  Set the default address translation function for all subsequently added memory spaces.
- void setDefaultAddressTranslateDelegate (MemoryAddressTranslateDelegate delegate=MemoryAddressTranslateDelegate())

  Set the default address translation function for all subsequently added memory spaces.
- template<typename T , IrisErrorCode(T::*)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult &) METHOD>
  void setDefaultAddressTranslateDelegate (T *instance)

  Set the default address translation function for all subsequently added memory spaces.
- template<IrisErrorCode(*)(EventStream *&, const EventSourceInfo &, const std::vector< std::string > &) FUNC>
  void setDefaultEsCreateDelegate ()

  Set the delegate that helps to create a new event stream for the simulation-specific event.
- void setDefaultEsCreateDelegate (EventStreamCreateDelegate delegate)

  Set the delegate that helps to create a new event stream for the simulation-specific event.
- template<typename T , IrisErrorCode(T::*)(EventStream *&, const EventSourceInfo &, const std::vector< std::string > &) METHOD>
  void setDefaultEsCreateDelegate (T *instance)

  Set the delegate that helps to create a new event stream for the simulation-specific event.
- template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std::vector< std::string > &, IrisValueMap &) FUNC>
  void setDefaultGetMemorySidebandInfoDelegate ()

  Set the default sideband info function for all subsequently added memory spaces.
- void setDefaultGetMemorySidebandInfoDelegate (MemoryGetSidebandInfoDelegate delegate)

  Set the default sideband info function for all subsequently added memory spaces.
- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std::vector< std::string > &, IrisValueMap &) METHOD>
  void setDefaultGetMemorySidebandInfoDelegate (T *instance)

  Set the default sideband info function for all subsequently added memory spaces.
- template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) FUNC>
  void setDefaultMemoryReadDelegate ()

  Set the default read function for all subsequently added memory spaces.
- void setDefaultMemoryReadDelegate (MemoryReadDelegate delegate=MemoryReadDelegate())

  Set the default read function for all subsequently added memory spaces.
- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) METHOD>
  void setDefaultMemoryReadDelegate (T *instance)

  Set the default read function for all subsequently added memory spaces.
- template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) FUNC>
  void setDefaultMemoryWriteDelegate ()

  Set default write function for all subsequently added memory spaces.
- void setDefaultMemoryWriteDelegate (MemoryWriteDelegate delegate=MemoryWriteDelegate())

  Set the default write function for all subsequently added memory spaces.

- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) METHOD>
  void setDefaultMemoryWriteDelegate (T *instance)

  Set the default write function for all subsequently added memory spaces.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) READER, IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &) WRITER>
  void setDefaultResourceDelegates (T *instance)

  Set both read and write resource delegates if they are defined in the same class.
- template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
  void setDefaultResourceReadDelegate ()

  Set default read access function for all subsequently added resources.
- void setDefaultResourceReadDelegate (ResourceReadDelegate delegate=ResourceReadDelegate())

  Set default read access function for all subsequently added resources.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
  void setDefaultResourceReadDelegate (T *instance)

  Set default read access function for all subsequently added resources.
- template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
  void setDefaultResourceWriteDelegate ()

  Set default write access function for all subsequently added resources.
- void setDefaultResourceWriteDelegate (ResourceWriteDelegate delegate=ResourceWriteDelegate())

  Set default write access function for all subsequently added resources.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &) METHOD>
  void setDefaultResourceWriteDelegate (T *instance)

  Set default write access function for all subsequently added resources.
- template<IrisErrorCode(*)(const TableInfo &, uint64_t, uint64_t, TableReadResult &) FUNC>
  void setDefaultTableReadDelegate ()

  Set the default table read function for all subsequently added tables.
- template<typename T , IrisErrorCode(T::*)(const TableInfo &, uint64_t, uint64_t, TableReadResult &) METHOD>
  void setDefaultTableReadDelegate (T *instance)

  Set the default table read function for all subsequently added tables.
- void setDefaultTableReadDelegate (TableReadDelegate delegate=TableReadDelegate())

  Set the default table read function for all subsequently added tables.
- template<IrisErrorCode(*)(const TableInfo &, const TableRecords &, TableWriteResult &) FUNC>
  void setDefaultTableWriteDelegate ()

  Set the default table write function for all subsequently added tables.
- template<typename T , IrisErrorCode(T::*)(const TableInfo &, const TableRecords &, TableWriteResult &) METHOD>
  void setDefaultTableWriteDelegate (T *instance)

  Set the default table write function for all subsequently added tables.
- void setDefaultTableWriteDelegate (TableWriteDelegate delegate=TableWriteDelegate())

  Set the default table write function for all subsequently added tables.
- template<IrisErrorCode(*)(bool &) FUNC>
  void setExecutionStateGetDelegate ()

  Set the delegate to get the execution state for this instance.
- void setExecutionStateGetDelegate (PerInstanceExecutionStateGetDelegate delegate)

  Set the delegate to get the execution state for this instance.
- template<typename T , IrisErrorCode(T::*)(bool &) METHOD>
  void setExecutionStateGetDelegate (T *instance)

  Set the delegate to get the execution state for this instance.
- template<IrisErrorCode(*)(bool) FUNC>
  void setExecutionStateSetDelegate ()

  Set the delegate to set the execution state for this instance.
- void setExecutionStateSetDelegate (PerInstanceExecutionStateSetDelegate delegate=PerInstanceExecutionStateSetDelegate

  Set the delegate to set the execution state for this instance.

- template<typename T , IrisErrorCode(T::*)(bool) METHOD>
  void setExecutionStateSetDelegate (T *instance)

  Set the delegate to set the execution state for this instance.
- void setHandleBreakpointHitsDelegate (std::function< IrisErrorCode(const BreakpointHitInfos &hitBpts)> delegate)

  Set the delegate that is called when a breakpoint is hit.
- template<IrisErrorCode(*)(const std::vector< uint8_t > &) FUNC>
  void setLoadImageDataDelegate ()

  Set the delegate to load an image from the data provided.
- void setLoadImageDataDelegate (ImageLoadDataDelegate delegate=ImageLoadDataDelegate())

  Set the delegate to load an image from the data provided.
- template<typename T , IrisErrorCode(T::*)(const std::vector< uint8_t > &) METHOD>
  void setLoadImageDataDelegate (T *instance)

  Set the delegate to load an image from the data provided.
- template<IrisErrorCode(*)(const std::string &) FUNC>
  void setLoadImageFileDelegate ()

  Set the delegate to load an image from a file.
- void setLoadImageFileDelegate (ImageLoadFileDelegate delegate=ImageLoadFileDelegate())

  Set the delegate to load an image from a file.
- template<typename T , IrisErrorCode(T::*)(const std::string &) METHOD>
  void setLoadImageFileDelegate (T *instance)

  Set the delegate to load an image from a file.
- void setNextSubRscId (uint64_t nextSubRscId)

  Set the rscId that will be used for the next resource to be added.
- void setPropertyCanonicalMsnScheme (const std::string &canonicalMsnScheme)

  Set the memory.canonicalMsnScheme instance property.
- void setPropertyCanonicalRnScheme (const std::string &canonicalRnScheme)

  Set the register.canonicalRnScheme instance property.
- EventSourceBuilder setRegisterReadEvent (const std::string &name, const std::string &description=std::string())

  Add a new register read event source.
- EventSourceBuilder setRegisterReadEvent (const std::string &name, IrisRegisterEventEmitterBase &event_emitter)

  Add a new register read event source.
- EventSourceBuilder setRegisterUpdateEvent (const std::string &name, const std::string &description=std::string())

  Add a new register update event source.
- EventSourceBuilder setRegisterUpdateEvent (const std::string &name, IrisRegisterEventEmitterBase &event_emitter)

  Add a new register update event source.
- template<IrisErrorCode(*)(uint64_t &, const std::string &) FUNC>
  void setRemainingStepGetDelegate ()

  Set the delegate to get the remaining steps for this instance.
- void setRemainingStepGetDelegate (RemainingStepGetDelegate delegate)

  Set the delegate to get the remaining steps for this instance.
- template<typename T , IrisErrorCode(T::*)(uint64_t &, const std::string &) METHOD>
  void setRemainingStepGetDelegate (T *instance)

  Set the delegate to get the remaining steps for this instance.
- template<IrisErrorCode(*)(uint64_t, const std::string &) FUNC>
  void setRemainingStepSetDelegate ()

  Set the delegate to set the remaining steps for this instance.
- void setRemainingStepSetDelegate (RemainingStepSetDelegate delegate=RemainingStepSetDelegate())

Set the delegate to set the remaining steps for this instance.

- template<typename T , IrisErrorCode(T::*)(uint64_t, const std::string &) METHOD>
  void setRemainingStepSetDelegate (T *instance)

  Set the delegate to set the remaining steps for this instance.
- template<IrisErrorCode(*)(uint64_t &, const std::string &) FUNC>
  void setStepCountGetDelegate ()

  Set the delegate to get the step count for this instance.
- void setStepCountGetDelegate (StepCountGetDelegate delegate=StepCountGetDelegate())

  Set the delegate to get the step count for this instance.
- template<typename T , IrisErrorCode(T::*)(uint64_t &, const std::string &) METHOD>
  void setStepCountGetDelegate (T *instance)

  Set the delegate to get the step count for this instance.
- void setTag (ResourceId rscId, const std::string &tag)

  Set a tag for a specific resource.

- void setGetCurrentDisassemblyModeDelegate (GetCurrentDisassemblyModeDelegate delegate)

  disass_apis IrisInstanceBuilder disassembler APIs
- template<typename T , IrisErrorCode(T::*)(std::string &) METHOD>
  void **setGetCurrentDisassemblyModeDelegate** (T *instance)
- void **setGetDisassemblyDelegate** (std::function< IrisErrorCode(GetDisassemblyArgs &)> delegate)

  Set the delegate to get the disassembly of a chunk of memory.
- void **setDisassembleOpcodeDelegate** (DisassembleOpcodeDelegate delegate)

  Set the delegate to get the disassembly of Opcode.
- template<typename T , IrisErrorCode(T::*)(const std::vector< uint64_t > &, uint64_t, const std::string &, DisassembleContext &, DisassemblyLine &) METHOD>
  void **setDisassembleOpcodeDelegate** (T *instance)
- template<IrisErrorCode(*)(const std::vector< uint64_t > &, uint64_t, const std::string &, DisassembleContext &, DisassemblyLine &) FUNC>
  void **setDisassembleOpcodeDelegate** ()
- void **addDisassemblyMode** (const std::string &name, const std::string &description)

  Add a disassembly mode.

- void setDbgStateSetRequestDelegate (DebuggableStateSetRequestDelegate delegate=DebuggableStateSetRequestDelegate

  debuggable_state_apis IrisInstanceBuilder debuggable state APIs
- template<typename T , IrisErrorCode(T::*)(bool) METHOD>
  void setDbgStateSetRequestDelegate (T *instance)

  Set the delegate to set the debuggable state request flag for this instance.
- template<IrisErrorCode(*)(bool) FUNC>
  void setDbgStateSetRequestDelegate ()

  Set the delegate to set the debuggable state request flag for this instance.
- void setDbgStateGetAcknowledgeDelegate (DebuggableStateGetAcknowledgeDelegate delegate=DebuggableStateGetAckno

  Set the delegate to get the debuggable state acknowledge flag for this instance.
- template<typename T , IrisErrorCode(T::*)(bool &) METHOD>
  void setDbgStateGetAcknowledgeDelegate (T *instance)

  Set the delegate to get the debuggable state acknowledge flag for this instance.
- template<IrisErrorCode(*)(bool &) FUNC>
  void setDbgStateGetAcknowledgeDelegate ()

  Set the delegate to get the debuggable state acknowledge flag for this instance.
- template<typename T , IrisErrorCode(T::*)(bool) SET_REQUEST, IrisErrorCode(T::*)(bool &) GET_ACKNOWLEDGE>
  void setDbgStateDelegates (T *instance)

Set both the debuggable state delegates.

- void **setCheckpointSaveDelegate** (CheckpointSaveDelegate delegate=CheckpointSaveDelegate())

  Delegates for checkpointing.

- template<typename T , IrisErrorCode(T::*)(const std::string &) METHOD>
  void **setCheckpointSaveDelegate** (T *instance)
- void **setCheckpointRestoreDelegate** (CheckpointRestoreDelegate delegate=CheckpointRestoreDelegate())
- template<typename T , IrisErrorCode(T::*)(const std::string &) METHOD>
  void **setCheckpointRestoreDelegate** (T *instance)

- SemihostingManager enableSemihostingAndGetManager ()

  Enable semihosting functionality for this instance and get a manager object to make use of it.

- bool hasAnyBreakpointSetOrTraceEnabled ()

  Check if there is any breakpoint set or if any trace is enabled for this instance.

8.19.1 Detailed Description

Builder interface to populate an IrisInstance with registers, memory etc.
See DummyComponent.h for a working example.

8.19.2 Constructor & Destructor Documentation

8.19.2.1 IrisInstanceBuilder()

```
iris::IrisInstanceBuilder::IrisInstanceBuilder (
            IrisInstance * iris_instance )
```

Construct an IrisInstanceBuilder for an Iris instance.

Parameters

iris_instance	The instance to build.

8.19.3 Member Function Documentation

8.19.3.1 addTable()

```
TableBuilder iris::IrisInstanceBuilder::addTable (
            const std::string & name )  [inline]
```

Add metadata for one table.
Typical use pattern:

```
addTableInfo("name")
    .setDescription("description")
    .setMinIndex(...)
    .setMaxIndex(...)
    .setIndexFormatHint(...)
    .setFormatShort(...)
    .setFormatLong(...)
    .setReadDelegate(...)
    .setWriteDelegate(...)
    .addColumnInfo(...)
    .addColumnInfo(...)
    ...
```

Parameters

name	Name of the new table.

Returns

A TableBuilder object than can be used to set metadata for the new table.

8.19.3.2 enableSemihostingAndGetManager()

```
SemihostingManager iris::IrisInstanceBuilder::enableSemihostingAndGetManager ( )  [inline]
```

Enable semihosting functionality for this instance and get a manager object to make use of it.

Returns

A SemihostingManager object to manage semihosting functionality for this instance.

8.19.3.3 getRegisterEventEmitterMap()

```
RscIdEventEmitterMap iris::IrisInstanceBuilder::getRegisterEventEmitterMap ( )  [inline]
```

Returns event emitters associated with registers.

Returns

A map of resourceID to pair (read, update) of event emitters.

8.19.3.4 hasAnyBreakpointSetOrTraceEnabled()

```
bool iris::IrisInstanceBuilder::hasAnyBreakpointSetOrTraceEnabled ( )  [inline]
```

Check if there is any breakpoint set or if any trace is enabled for this instance.

Returns

true if any breakpoint is set or if any trace is enabled for this instance.

8.19.3.5 setDbgStateDelegates()

```
template<typename T , IrisErrorCode(T::*)(bool) SET_REQUEST, IrisErrorCode(T::*)(bool &) GET↩
_ACKNOWLEDGE>
void iris::IrisInstanceBuilder::setDbgStateDelegates (
            T * instance )  [inline]
```

Set both the debuggable state delegates.

Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode setRequestFlag(bool request_debuggable_state);
    iris::IrisErrorCode getAcknowledgeFlag(bool &debuggable_state_acknowledge);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDbgStateDelegates<MyClass,
                              &MyClass::setRequest,
                              &MyClass::getAcknowledgeFlag>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines both a debuggable state request set and a get acknowledge delegate method.
SET_REQUEST	A method of class T which is a debuggable state request set delegate.
GET_ACKNOWLEDGE	A method of class T which is a debuggable state get acknowledge delegate.

Parameters

instance	An instance of class T on which SET_REQUEST and GET_ACKNOWLEDGE should be called.

8.19.3.6 setDbgStateGetAcknowledgeDelegate() [1/3]

```
template<IrisErrorCode(*)(bool &) FUNC>
void iris::IrisInstanceBuilder::setDbgStateGetAcknowledgeDelegate ( )  [inline]
```

Set the delegate to get the debuggable state acknowledge flag for this instance.
Usage:

```
    iris::IrisErrorCode getAcknowledgeFlag(bool &debuggable_state_acknowledge);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDbgStateGetAcknowledgeDelegate<&getAcknowledgeFlag>();
```

Template Parameters

FUNC	Global function to call to get the debuggable state acknowledge flag.

8.19.3.7 setDbgStateGetAcknowledgeDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDbgStateGetAcknowledgeDelegate (
            DebuggableStateGetAcknowledgeDelegate delegate = DebuggableStateGetAcknowledgeDelegate()
)  [inline]
```

Set the delegate to get the debuggable state acknowledge flag for this instance.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getAcknowledgeFlag(bool &debuggable_state_acknowledge);
};
MyClass myInstanceOfMyClass;
iris::DebuggableStateGetAcknowledgeDelegate delegate =
    iris::DebuggableStateGetAcknowledgeDelegate::make<MyClass,
       &MyClass::getAcknowledgeFlag>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDbgStateGetAcknowledgeDelegate(delegate);
```

Parameters

delegate	Delegate object to call to get the debuggable state acknowledge flag.

8.19.3.8 setDbgStateGetAcknowledgeDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(bool &) METHOD>
void iris::IrisInstanceBuilder::setDbgStateGetAcknowledgeDelegate (
            T * instance )  [inline]
```

Set the delegate to get the debuggable state acknowledge flag for this instance.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getAcknowledgeFlag(bool &debuggable_state_acknowledge);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDbgStateGetAcknowledgeDelegate<MyClass, &MyClass::getAcknowledgeFlag>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a debuggable state get acknowledge delegate method.
METHOD	A method of class T which is a debuggable state get acknowledge delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

8.19.3.9 setDbgStateSetRequestDelegate() [1/3]

```
template<IrisErrorCode(*)(bool) FUNC>
void iris::IrisInstanceBuilder::setDbgStateSetRequestDelegate ( )  [inline]
```

Set the delegate to set the debuggable state request flag for this instance.
Usage:

```
iris::IrisErrorCode setRequestFlag(bool request_debuggable_state);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDbgStateSetRequestDelegate<&setRequestFlag>();
```

Template Parameters

FUNC	Global function to call to set the debuggable state request flag.

8.19.3.10 setDbgStateSetRequestDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setDbgStateSetRequestDelegate (
            DebuggableStateSetRequestDelegate delegate = DebuggableStateSetRequestDelegate()
)  [inline]
```

debuggable_state_apis IrisInstanceBuilder debuggable state APIs
Set the delegate to set the debuggable state request flag for this instance.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode setRequestFlag(bool request_debuggable_state);
};
MyClass myInstanceOfMyClass;
iris::DebuggableStateSetRequestDelegate delegate =
    iris::DebuggableStateSetRequestDelegate::make<MyClass, &MyClass::setRequestFlag>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDbgStateSetRequestDelegate(delegate);
```

Parameters

delegate	Delegate object to call to set the debuggable state request flag.

8.19.3.11 setDbgStateSetRequestDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(bool) METHOD>
void iris::IrisInstanceBuilder::setDbgStateSetRequestDelegate (
            T * instance )  [inline]
```

Set the delegate to set the debuggable state request flag for this instance.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode setRequestFlag(bool request_debuggable_state);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDbgStateSetRequestDelegate<MyClass, &MyClass::setRequestFlag>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a debuggable state request set delegate method.
METHOD	A method of class T which is a debuggable state request set delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

8.19.3.12 setDefaultTableReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const TableInfo &, uint64_t, uint64_t, TableReadResult &) FUNC>
void iris::IrisInstanceBuilder::setDefaultTableReadDelegate ( )  [inline]
```

Set the default table read function for all subsequently added tables.

Tables that do not explicitly override the access function using
`addTable(...).setReadDelegate(...)`
will use this delegate.

Usage:

```
iris::IrisErrorCode readTable(const iris::TableInfo &tableInfo, uint64_t index,
                              uint64_t count, iris::TableReadResult &result);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultTableReadDelegate<&readTable>();
builder->addTable(...); // Uses readTable
```

Template Parameters

FUNC	Global function to call to read a table.

8.19.3.13 setDefaultTableReadDelegate() [2/3]

```
template<typename T , IrisErrorCode(T::*)(const TableInfo &, uint64_t, uint64_t, TableRead↩
Result &) METHOD>
void iris::IrisInstanceBuilder::setDefaultTableReadDelegate (
             T * instance )  [inline]
```

Set the default table read function for all subsequently added tables.

Tables that do not explicitly override the access function using
`addTable(...).setReadDelegate(...)`
will use this delegate.

Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode readTable(const iris::TableInfo &tableInfo, uint64_t index,
                                  uint64_t count, iris::TableReadResult &result);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultTableReadDelegate<MyClass, &MyClass::readTable>(&myInstanceOfMyClass);
builder->addTable(...); // Uses readTable
```

Template Parameters

T	Class that defines a table read delegate method.
METHOD	A method of class T which is a table read delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

8.19.3.14 setDefaultTableReadDelegate() [3/3]

```
void iris::IrisInstanceBuilder::setDefaultTableReadDelegate (
            TableReadDelegate delegate = TableReadDelegate() )  [inline]
```

Set the default table read function for all subsequently added tables.
Tables that do not explicitly override the access function using
`addTable(...).setReadDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode readTable(const iris::TableInfo &tableInfo, uint64_t index,
                                  uint64_t count, iris::TableReadResult &result);
};
MyClass myInstanceOfMyClass;
iris::TableReadDelegate delegate =
    iris::TableReadDelegate::make<MyClass, &MyClass::readTable>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultTableReadDelegate(delegate);
builder->addTable(...); // Uses readTable
```

Parameters

delegate	Delegate object to call to read a table.

8.19.3.15 setDefaultTableWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const TableInfo &, const TableRecords &, TableWriteResult &) FUNC>
void iris::IrisInstanceBuilder::setDefaultTableWriteDelegate ( )  [inline]
```

Set the default table write function for all subsequently added tables.
Tables that do not explicitly override the access function using
`addTable(...).setWriteDelegate(...)`
will use this delegate.
Usage:

```
iris::IrisErrorCode writeTable(const iris::TableInfo &tableInfo,
                               const iris::TableRecords &records,
                               iris::TableWriteResult &result);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultTableWriteDelegate<&writeTable>();
builder->addTable(...); // Uses writeTable
```

Template Parameters

FUNC	Global function to call to write a table.

8.19.3.16 setDefaultTableWriteDelegate() [2/3]

```
template<typename T , IrisErrorCode(T::*)(const TableInfo &, const TableRecords &, Table↩
WriteResult &) METHOD>
void iris::IrisInstanceBuilder::setDefaultTableWriteDelegate (
            T * instance )  [inline]
```

Set the default table write function for all subsequently added tables.
Tables that do not explicitly override the access function using
`addTable(...).setWriteDelegate(...)`
will use this delegate.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode writeTable(const iris::TableInfo &tableInfo,
                                   const iris::TableRecords &records,
                                   iris::TableWriteResult &result);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultTableWriteDelegate<MyClass, &MyClass::writeTable>(&myInstanceOfMyClass);
builder->addTable(...); // Uses writeTable
```

Template Parameters

T	Class that defines a table write delegate method.
METHOD	A method of class T which is a table write delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

8.19.3.17 setDefaultTableWriteDelegate() [3/3]

```
void iris::IrisInstanceBuilder::setDefaultTableWriteDelegate (
            TableWriteDelegate delegate = TableWriteDelegate() )  [inline]
```

Set the default table write function for all subsequently added tables.
Tables that do not explicitly override the access function using
`addTable(...).setWriteDelegate(...)`
will use this delegate.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩
not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode writeTable(const iris::TableInfo &tableInfo,
                                   const iris::TableRecords &records,
                                   iris::TableWriteResult &result);
};
MyClass myInstanceOfMyClass;
iris::TableWriteDelegate delegate =
    iris::TableWriteDelegate::make<MyClass, &MyClass::writeTable>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setDefaultTableWriteDelegate(delegate);
builder->addTable(...); // Uses writeTable
```

Parameters

delegate	Delegate object to call to write a table.

8.19.3.18 setExecutionStateGetDelegate() [1/3]

```
template<IrisErrorCode(*)(bool &) FUNC>
void iris::IrisInstanceBuilder::setExecutionStateGetDelegate ( )  [inline]
```

Set the delegate to get the execution state for this instance.

Usage:

```
iris::IrisErrorCode getState(bool &execution_enabled);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setExecutionStateGetDelegate<&getState>();
```

Template Parameters

FUNC	Global function to call to get the execution state.

8.19.3.19 setExecutionStateGetDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setExecutionStateGetDelegate (
            PerInstanceExecutionStateGetDelegate delegate )  [inline]
```

Set the delegate to get the execution state for this instance.

Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.

Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getState(bool &execution_enabled);
};
MyClass myInstanceOfMyClass;
iris::PerInstanceExecutionStateGetDelegate delegate =
    iris::PerInstanceExecutionStateGetDelegate::make<MyClass, &MyClass::getState>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setExecutionStateGetDelegate(delegate);
```

Parameters

delegate	Delegate object to call to get the execution state.

8.19.3.20 setExecutionStateGetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(bool &) METHOD>
void iris::IrisInstanceBuilder::setExecutionStateGetDelegate (
            T * instance )  [inline]
```

Set the delegate to get the execution state for this instance.

Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode getState(bool &execution_enabled);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setExecutionStateGetDelegate<MyClass, &MyClass::getState>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a get execution state delegate method.
METHOD	A method of class T which is a get execution state delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

8.19.3.21 setExecutionStateSetDelegate() [1/3]

```
template<IrisErrorCode(*)(bool) FUNC>
void iris::IrisInstanceBuilder::setExecutionStateSetDelegate ( )  [inline]
```

Set the delegate to set the execution state for this instance.
Usage:

```
iris::IrisErrorCode setState(bool enable_execution);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setExecutionStateSetDelegate<&setState>();
```

Template Parameters

FUNC	Global function to call to set the execution state.

8.19.3.22 setExecutionStateSetDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setExecutionStateSetDelegate (
            PerInstanceExecutionStateSetDelegate delegate = PerInstanceExecutionStateSetDelegate()
)  [inline]
```

Set the delegate to set the execution state for this instance.
Passing an empty delegate (the default argument) restores the default implementation which always returns E_↩ not_implemented for all requests.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode setState(bool enable_execution);
};
MyClass myInstanceOfMyClass;
iris::PerInstanceExecutionStateSetDelegate delegate =
    iris::PerInstanceExecutionStateSetDelegate::make<MyClass, &MyClass::setState>(&myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setExecutionStateSetDelegate(delegate);
```

Parameters

delegate	Delegate object to call to set the execution state.

8.19.3.23 setExecutionStateSetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(bool) METHOD>
void iris::IrisInstanceBuilder::setExecutionStateSetDelegate (
            T * instance )  [inline]
```

Set the delegate to set the execution state for this instance.
Usage:

```
class MyClass
{
    ...
    iris::IrisErrorCode setState(bool enable_execution);
};
MyClass myInstanceOfMyClass;
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setExecutionStateSetDelegate<MyClass, &MyClass::setState>(&myInstanceOfMyClass);
```

Template Parameters

T	Class that defines a set execution state delegate method.
METHOD	A method of class T which is a set execution state delegate.

Parameters

instance	An instance of class T on which METHOD should be called.

8.19.3.24 setGetCurrentDisassemblyModeDelegate()

```
void iris::IrisInstanceBuilder::setGetCurrentDisassemblyModeDelegate (
            GetCurrentDisassemblyModeDelegate delegate )  [inline]
```

disass_apis IrisInstanceBuilder disassembler APIs
Set the delegates to get the current disassembly mode
The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.20 iris::IrisInstanceCheckpoint Class Reference

Checkpoint add-on for IrisInstance.
`#include <IrisInstanceCheckpoint.h>`

Public Member Functions

- void attachTo (IrisInstance *iris_instance_)

  Attach this IrisInstance add-on to a specific IrisInstance.
- **IrisInstanceCheckpoint** (IrisInstance *iris_instance=nullptr)
- void setCheckpointRestoreDelegate (CheckpointRestoreDelegate delegate)

  Set checkpoint restore delegate for all checkpoints related to this instance.
- void setCheckpointSaveDelegate (CheckpointSaveDelegate delegate)

  Set checkpoint save delegate for all checkpoints related to this instance.

8.20.1 Detailed Description

Checkpoint add-on for IrisInstance.

8.20.2 Member Function Documentation

8.20.2.1 attachTo()

```
void iris::IrisInstanceCheckpoint::attachTo (
            IrisInstance * iris_instance_ )
```

Attach this IrisInstance add-on to a specific IrisInstance.
Only use this method if nullptr was passed to the constructor.

Parameters

*iris_↩ instance_*	The IrisInstance to attach to.

8.20.2.2 setCheckpointRestoreDelegate()

```
void iris::IrisInstanceCheckpoint::setCheckpointRestoreDelegate (
            CheckpointRestoreDelegate delegate )
```

Set checkpoint restore delegate for all checkpoints related to this instance.

Parameters

delegate	A CheckpointRestoreDelegate to call when restoring a checkpoint.

8.20.2.3 setCheckpointSaveDelegate()

```
void iris::IrisInstanceCheckpoint::setCheckpointSaveDelegate (
            CheckpointSaveDelegate delegate )
```

Set checkpoint save delegate for all checkpoints related to this instance.

Parameters

delegate	A CheckpointSaveDelegate to call when saving a checkpoint.

The documentation for this class was generated from the following file:

- IrisInstanceCheckpoint.h

8.21 iris::IrisInstanceDebuggableState Class Reference

Debuggable-state add-on for IrisInstance.

`#include <IrisInstanceDebuggableState.h>`

Public Member Functions

- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- **IrisInstanceDebuggableState** (IrisInstance *iris_instance=nullptr)
- void setGetAcknowledgeDelegate (DebuggableStateGetAcknowledgeDelegate delegate)

  Set the get acknowledge flag delegate.
- void setSetRequestDelegate (DebuggableStateSetRequestDelegate delegate)

  Set the set request flag delegate.

8.21.1 Detailed Description

Debuggable-state add-on for IrisInstance.

8.21.2 Member Function Documentation

8.21.2.1 attachTo()

```
void iris::IrisInstanceDebuggableState::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.21.2.2 setGetAcknowledgeDelegate()

```
void iris::IrisInstanceDebuggableState::setGetAcknowledgeDelegate (
            DebuggableStateGetAcknowledgeDelegate delegate )  [inline]
```

Set the get acknowledge flag delegate.

Parameters

delegate	Delegate that will be called to get the debuggable-state acknowledge flag.

8.21.2.3 setSetRequestDelegate()

```
void iris::IrisInstanceDebuggableState::setSetRequestDelegate (
            DebuggableStateSetRequestDelegate delegate )  [inline]
```

Set the set request flag delegate.

Parameters

delegate	Delegate that will be called to set or clear the debuggable-state request flag.

The documentation for this class was generated from the following file:

- IrisInstanceDebuggableState.h

8.22 iris::IrisInstanceDisassembler Class Reference

Disassembler add-on for IrisInstance.

`#include <IrisInstanceDisassembler.h>`

Public Member Functions

- void addDisassemblyMode (const std::string &name, const std::string &description)

  Add a disassembly mode.
- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- IrisInstanceDisassembler (IrisInstance *irisInstance=nullptr)

  Construct an IrisInstanceDisassembler.
- void setDisassembleOpcodeDelegate (DisassembleOpcodeDelegate delegate)

  Set the delegate to get the disassembly of Opcode.
- void setGetCurrentModeDelegate (GetCurrentDisassemblyModeDelegate delegate)

  Set the delegate to get the current disassembly mode.
- void setGetDisassemblyDelegate (std::function< IrisErrorCode(GetDisassemblyArgs &)> delegate)

  Set the delegate to get the disassembly of a chunk of memory.

8.22.1 Detailed Description

Disassembler add-on for IrisInstance.
This class is used by instances that want to support disassembly functionality.
It implements all Iris disassembler*() functions.
Example usage:

```
irisInstanceDisassembler = new iris::IrisInstanceDisassembler(irisInstance);
irisInstanceDisassembler->setGetCurrentModeDelegate(dasmCurrentModeGetDel);  // Get the current disassembly
       mode
irisInstanceDisassembler->setGetDisassemblyDelegate(dasmDisassemblyGetDel);  // Get the disassembly of a
       chunk of memory
```

```
irisInstanceDisassembler->setDisassembleOpcodeDelegate(dasmOpcodeDasmGetDel);    // Disassemble specific
      opcode
```

See DummyComponent.h for a working example.
The documentation for this class was generated from the following file:

- IrisInstanceDisassembler.h

8.23 iris::IrisInstanceEvent Class Reference

Event add-on for IrisInstance.
`#include <IrisInstanceEvent.h>`

Classes

- struct EventSourceInfoAndDelegate

  Contains the metadata and delegates for a single EventSource.
- struct ProxyEventInfo

  Contains information for a single proxy EventSource.

Public Member Functions

- uint64_t addEventSource (const EventSourceInfoAndDelegate &info)

  Add metadata for an event source.
- EventSourceInfoAndDelegate & addEventSource (const std::string &name, bool isHidden=false)

  Add metadata for an event source.
- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstanceEvent add-on to a specific IrisInstance.
- void deleteEventSource (const std::string &eventName)

  Delete metadata for an event source.
- void destroyAllEventStreams ()

  Destroy all event streams.
- bool destroyEventStream (EventStreamId esId)

  Destroy event stream (direct variant of eventStream_destroy()).
- EventSourceInfoAndDelegate & enhanceEventSource (const std::string &name)

  Enhance existing event source.
- void eventBufferClear (EventBufferId evBufId)

  Clear event buffer.
- const uint64_t * eventBufferGetSyncStepResponse (EventBufferId evBufId, RequestId requestId)

  Get response to step_syncStep(), containing event data.
- const EventSourceInfo * getEventSourceInfo (EventSourceId evSrcId) const

  Get EventSourceInfo for EventSourceId.
- bool hasEventSource (const std::string &eventName)

  Check if event source already exists.
- bool **hasEventStreams** () const
- IrisInstanceEvent (IrisInstance *irisInstance=nullptr)

  Construct an IrisInstanceEvent add-on.
- bool isValidEvBufId (EventBufferId evBufId) const

  Check whether event buffer id is valid.
- void renameEventSource (const std::string &name, const std::string &newName)

  Rename existing event source.
- void setDefaultEsCreateDelegate (EventStreamCreateDelegate delegate)

  Set the default delegate for creating EventStreams for the attached instance.

Friends

- struct **EventBuffer**

8.23.1 Detailed Description

Event add-on for IrisInstance.
This class is used by instances to support event functionality. Generally, there are two kinds of event sources:

- Iris-specific event sources. These are defined in the Iris spec, for example `IRIS_BREAKPOINT_HIT` and `IRIS_SIMULATION_TIME_EVENT`.
- Simulation-specific event sources. These are not defined in the Iris spec. They could be quite different for different simulations or instances. For example `INST` (every instruction executed).

This class implements all Iris event*() functions. It maintains event source information that is added by addEventSource() and exposed by event_getEventSources() or event_getEventSource(). This class maintains all event streams. Iris-specific event streams are created by this add-on. Simulation-specific event streams are created by a delegate, which could be different for different simulations or instances.

8.23.2 Constructor & Destructor Documentation

8.23.2.1 IrisInstanceEvent()

```
iris::IrisInstanceEvent::IrisInstanceEvent (
            IrisInstance * irisInstance = nullptr )
```

Construct an IrisInstanceEvent add-on.

Parameters

irisInstance	The IrisInstance to which to attach this add-on.

8.23.3 Member Function Documentation

8.23.3.1 addEventSource() [1/2]

```
uint64_t iris::IrisInstanceEvent::addEventSource (
            const EventSourceInfoAndDelegate & info )
```

Add metadata for an event source.

Parameters

info	The metadata and event-specific delegates (if applicable) for a new event to add.

Returns

The evSrcId of the newly added event source.

8.23.3.2 addEventSource() [2/2]

```
EventSourceInfoAndDelegate & iris::IrisInstanceEvent::addEventSource (
            const std::string & name,
            bool isHidden = false )
```

Add metadata for an event source.

Parameters

name	The name of the event source.
isHidden	If true, this event source is hidden. The EventSourceInfo is not included in the list of event sources returned by event_getEventSources() but can still be accessed by event_getEventSource() if the client knows the name of the hidden event.

Returns

A reference to an object which keeps the metadata and event-specific delegates (if applicable) for this event. The reference is valid until the next call to addEventSource().

8.23.3.3 attachTo()

```
void iris::IrisInstanceEvent::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstanceEvent add-on to a specific IrisInstance.
This should only be used if no instance was attached when this object was constructed.

Parameters

irisInstance	The IrisInstance to which to attach this add-on.

8.23.3.4 deleteEventSource()

```
void iris::IrisInstanceEvent::deleteEventSource (
            const std::string & eventName )
```

Delete metadata for an event source.

Parameters

eventName	The name of the event source.

8.23.3.5 destroyAllEventStreams()

```
void iris::IrisInstanceEvent::destroyAllEventStreams ( )
```

Destroy all event streams.
All event streams are always automatically destroyed when IrisInstance (and so IrisInstanceEvent) is destroyed. This function allows to destroy all event streams to be destroyed before IrisInstance.
This is necessary when the event streams use other resources (like MTI traces) which go out of scope before IrisInstance does.

8.23.3.6 destroyEventStream()

```
bool iris::IrisInstanceEvent::destroyEventStream (
            EventStreamId esId )
```

Destroy event stream (direct variant of eventStream_destroy()).
If the event stream id is valid, disable and destroy the event stream. If disabling the event stream fails this is silently ignored (unlike eventStream_destroy()).

Returns

True if the event stream id was valid, else false.

8.23.3.7 enhanceEventSource()

```
EventSourceInfoAndDelegate & iris::IrisInstanceEvent::enhanceEventSource (
            const std::string & name )
```

Enhance existing event source.

Parameters

name	The name of the event source.

Returns

A reference to an object which keeps the metadata and event-specific delegates (if applicable) for this event. The reference is valid until the next call to addEventSource().

8.23.3.8 eventBufferClear()

```
void iris::IrisInstanceEvent::eventBufferClear (
            EventBufferId evBufId )
```

Clear event buffer.
This is separate from eventBufferGetSyncStepResponse() so the message writer can be used to send the message without taking an unnecessary copy.

Parameters

evBufId	The event buffer which is to be cleared.

8.23.3.9 eventBufferGetSyncStepResponse()

```
const uint64_t * iris::IrisInstanceEvent::eventBufferGetSyncStepResponse (
            EventBufferId evBufId,
            RequestId requestId )
```

Get response to step_syncStep(), containing event data.

Parameters

evBufId	The data of this event buffer is returned. This is set beforehand with step_syncStepSetup().
requestId	This is the request id of the original step_syncStep() for which this function generates the answer.

Returns

Response message to step_syncStep() call, containing the event data.

8.23.3.10 getEventSourceInfo()

```
const EventSourceInfo * iris::IrisInstanceEvent::getEventSourceInfo (
            EventSourceId evSrcId ) const
```

Get EventSourceInfo for EventSourceId.
Returns nullptr if the event source id is not found.

8.23.3.11 hasEventSource()

```
bool iris::IrisInstanceEvent::hasEventSource (
            const std::string & eventName )
```

Check if event source already exists.

Parameters

eventName	The name of the event source.

Returns

True iff event source already exists.

8.23.3.12 isValidEvBufId()

```
bool iris::IrisInstanceEvent::isValidEvBufId (
            EventBufferId evBufId ) const
```

Check whether event buffer id is valid.
This function is use to validate event buffer ids.

Returns

Returns true iff evBufId is a valid event buffer id.

8.23.3.13 renameEventSource()

```
void iris::IrisInstanceEvent::renameEventSource (
            const std::string & name,
            const std::string & newName )
```

Rename existing event source.
If an event source "newName" already exists, it is deleted/overwritten.

Parameters

name	The old name of the event source.
newName	The new name of the event source.

8.23.3.14 setDefaultEsCreateDelegate()

```
void iris::IrisInstanceEvent::setDefaultEsCreateDelegate (
            EventStreamCreateDelegate delegate )
```

Set the default delegate for creating EventStreams for the attached instance.

Parameters

delegate	A delegate that will be called to create an event stream for event sources in the attached instance that have not set an event source-specific delegate.

The documentation for this class was generated from the following file:

- IrisInstanceEvent.h

8.24 iris::IrisInstanceFactoryBuilder Class Reference

A builder class to construct instantiation parameter metadata.
`#include <IrisInstanceFactoryBuilder.h>`
Inherited by iris::IrisPluginFactoryBuilder.

Public Member Functions

- IrisParameterBuilder **addBooleanParameter** (const std::string &name, const std::string &description)
- IrisParameterBuilder addBoolParameter (const std::string &name, const std::string &description)

  Add a new boolean parameter.
- IrisParameterBuilder **addHiddenBooleanParameter** (const std::string &name, const std::string &description)
- IrisParameterBuilder addHiddenBoolParameter (const std::string &name, const std::string &description)

  Add a new hidden boolean parameter.
- IrisParameterBuilder addHiddenParameter (const std::string &name, uint64_t bitWidth, const std::string &description)

  Add a new hidden numeric parameter.
- IrisParameterBuilder addHiddenStringParameter (const std::string &name, const std::string &description)

  Add a new hidden string parameter.
- IrisParameterBuilder addParameter (const std::string &name, uint64_t bitWidth, const std::string &description)

  Add a new numeric parameter.
- IrisParameterBuilder addStringParameter (const std::string &name, const std::string &description)

  Add a new string parameter.
- const std::vector< ResourceInfo > & getHiddenParameterInfo () const

  Get all ResourceInfo for hidden parameters.
- const std::vector< ResourceInfo > & getParameterInfo () const

  Get all ResourceInfo for non-hidden parameters.
- IrisInstanceFactoryBuilder (const std::string &prefix)

  Construct an IrisInstanceFactoryBuilder.

8.24.1 Detailed Description

A builder class to construct instantiation parameter metadata.

8.24.2 Constructor & Destructor Documentation

8.24.2.1 IrisInstanceFactoryBuilder()

```
iris::IrisInstanceFactoryBuilder::IrisInstanceFactoryBuilder (
            const std::string & prefix )  [inline]
```

Construct an IrisInstanceFactoryBuilder.

Parameters

prefix	All parameters added to this builder are prefixed with this string.

8.24.3 Member Function Documentation

8.24.3.1 addBoolParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addBoolParameter (
            const std::string & name,
            const std::string & description )  [inline]
```

Add a new boolean parameter.
Boolean parameters are numeric parameters with a bitWidth of 1 and "true" and "false" enum symbols.

Parameters

name	Name of the parameter.
description	Description of the parameter.

Returns

An IrisParameterBuilder object which can be used to set further metadata for this parameter. The object is valid until another parameter is added.

8.24.3.2 addHiddenBoolParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addHiddenBoolParameter (
            const std::string & name,
            const std::string & description )  [inline]
```

Add a new hidden boolean parameter.
Boolean parameters are numeric parameters with a bitWidth of 1 and "true" and "false" enum symbols.

Parameters

name	Name of the parameter.
description	Description of the parameter.

Returns

An IrisParameterBuilder object which can be used to set further metadata for this parameter. The object is valid until another parameter is added.

8.24.3.3 addHiddenParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addHiddenParameter (
            const std::string & name,
            uint64_t bitWidth,
            const std::string & description )  [inline]
```

Add a new hidden numeric parameter.

Parameters

name	Name of the parameter.
bitWidth	Width of the parameter in bits.
description	Description of the parameter.

Returns

An IrisParameterBuilder object which can be used to set further metadata for this parameter. The object is valid until another parameter is added.

8.24.3.4 addHiddenStringParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addHiddenStringParameter (
            const std::string & name,
            const std::string & description )  [inline]
```

Add a new hidden string parameter.

Parameters

name	Name of the parameter.
description	Description of the parameter.

Returns

An IrisParameterBuilder object which can be used to set further metadata for this parameter. The object is valid until another parameter is added.

8.24.3.5 addParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addParameter (
            const std::string & name,
            uint64_t bitWidth,
            const std::string & description )  [inline]
```

Add a new numeric parameter.

Parameters

name	Name of the parameter.
bitWidth	Width of the parameter in bits.
description	Description of the parameter.

Returns

An IrisParameterBuilder object which can be used to set further metadata for this parameter. The object is valid until another parameter is added.

8.24.3.6 addStringParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addStringParameter (
            const std::string & name,
            const std::string & description )  [inline]
```

Add a new string parameter.

Parameters

name	Name of the parameter.
description	Description of the parameter.

Returns

An IrisParameterBuilder object which can be used to set further metadata for this parameter. The object is valid until another parameter is added.

8.24.3.7 getHiddenParameterInfo()

```
const std::vector< ResourceInfo > & iris::IrisInstanceFactoryBuilder::getHiddenParameterInfo (
) const  [inline]
```

Get all ResourceInfo for hidden parameters.

Returns

A vector of ResourceInfo. Iterators for this vector are invalidated if a new hidden parameter is added.

8.24.3.8 getParameterInfo()

```
const std::vector< ResourceInfo > & iris::IrisInstanceFactoryBuilder::getParameterInfo ( )
const  [inline]
```

Get all ResourceInfo for non-hidden parameters.

Returns

A vector of ResourceInfo. Iterators for this vector are invalidated if a new non-hidden parameter is added.

The documentation for this class was generated from the following file:

- IrisInstanceFactoryBuilder.h

8.25 iris::IrisInstanceImage Class Reference

Image loading add-on for IrisInstance.

```
#include <IrisInstanceImage.h>
```

Public Member Functions

- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- IrisInstanceImage (IrisInstance *irisInstance=0)

  Construct a new IrisInstanceImage.
- void setLoadImageDataDelegate (ImageLoadDataDelegate delegate)

  Set image loading from (pushed/pulled) data delegate.
- void setLoadImageFileDelegate (ImageLoadFileDelegate delegate)

  Set image loading from file delegate.

Static Public Member Functions

- static IrisErrorCode readFileData (const std::string &fileName, std::vector< uint8_t > &data)

  Read file data into a uint8_t array.

8.25.1 Detailed Description

Image loading add-on for IrisInstance.
This class is used by instances to support image loading. It is also used by instances that want to use image_loadDataPull() to implement the image_loadDataRead() callback.
This class implements the Iris image*() functions. It maintains or implements two main things:

- Functions to load images:
  - From a file, by image_loadFile(), or from a data buffer, by image_loadData() or image_loadDataPull().
  - As raw data, by specifying rawAddr and rawSpaceId.
- Image meta information, which is exposed by image_getMetaInfoList() or cleared by image_clearMetaInfo↩ List().

See DummyComponent.h for a working example.

8.25.2 Constructor & Destructor Documentation

8.25.2.1 IrisInstanceImage()

```
iris::IrisInstanceImage::IrisInstanceImage (
            IrisInstance * irisInstance = 0 )
```

Construct a new IrisInstanceImage.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.25.3 Member Function Documentation

8.25.3.1 attachTo()

```
void iris::IrisInstanceImage::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.25.3.2 readFileData()

```
static IrisErrorCode iris::IrisInstanceImage::readFileData (
            const std::string & fileName,
            std::vector< uint8_t > & data )  [static]
```

Read file data into a uint8_t array.

Parameters

fileName	Name of the file to read.
data	A reference to a vector which is populated with the file contents.

Returns

An error code indicating success or failure.

8.25.3.3 setLoadImageDataDelegate()

```
void iris::IrisInstanceImage::setLoadImageDataDelegate (
            ImageLoadDataDelegate delegate )
```

Set image loading from (pushed/pulled) data delegate.

Parameters

delegate	The delegate that will be called to load an image from a data buffer.

8.25.3.4 setLoadImageFileDelegate()

```
void iris::IrisInstanceImage::setLoadImageFileDelegate (
            ImageLoadFileDelegate delegate )
```

Set image loading from file delegate.

Parameters

delegate	The delegate that will be called to load an image from a file.

The documentation for this class was generated from the following file:

- IrisInstanceImage.h

8.26 iris::IrisInstanceImage_Callback Class Reference

Image loading add-on for IrisInstance clients implementing image_loadDataRead().
`#include <IrisInstanceImage.h>`

Public Member Functions

- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- IrisInstanceImage_Callback (IrisInstance *irisInstance=0)

  Construct an IrisInstanceImage_Callback add-on.
- uint64_t openImage (const std::string &fileName)

  Open an image for reading.

Protected Member Functions

- void **impl_image_loadDataRead** (IrisReceivedRequest &request)

  Implementation of the Iris function image_loadDataRead().

8.26.1 Detailed Description

Image loading add-on for IrisInstance clients implementing image_loadDataRead().
This is used by instances that call the instances supporting image_loadDataPull().
This class maintains/implements:

- Iris image_loadDataRead() function.
- Image opening, data reading.
- Tags of images.

8.26.2 Constructor & Destructor Documentation

8.26.2.1 IrisInstanceImage_Callback()

```
iris::IrisInstanceImage_Callback::IrisInstanceImage_Callback (
            IrisInstance * irisInstance = 0 )
```

Construct an IrisInstanceImage_Callback add-on.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.26.3 Member Function Documentation

8.26.3.1 attachTo()

```
void iris::IrisInstanceImage_Callback::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.26.3.2 openImage()

```
uint64_t iris::IrisInstanceImage_Callback::openImage (
            const std::string & fileName )
```

Open an image for reading.

Parameters

fileName	File name of the image file to read.

Returns

An opaque tag number that is passed to image_loadDataRead() to identify the file to read from. This returns `iris::IRIS_UINT64_MAX` on failure to open the image.

The documentation for this class was generated from the following file:

- IrisInstanceImage.h

8.27 iris::IrisInstanceMemory Class Reference

Memory add-on for IrisInstance.

`#include <IrisInstanceMemory.h>`

Classes

- struct AddressTranslationInfoAndAccess

Contains static address translation information.

- struct SpaceInfoAndAccess

  Entry in 'spaceInfos'.

Public Member Functions

- AddressTranslationInfoAndAccess & addAddressTranslation (MemorySpaceId inSpaceId, MemorySpaceId outSpaceId, const std::string &description)

  Add one memory address translation as well as the translate interface.
- SpaceInfoAndAccess & addMemorySpace (const std::string &name)

  Add meta information for one memory space.
- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- IrisInstanceMemory (IrisInstance *irisInstance=0)

  Construct an IrisInstanceMemory.
- void setDefaultGetSidebandInfoDelegate (MemoryGetSidebandInfoDelegate delegate=MemoryGetSidebandInfoDelegate())

  Set the default delegate to retrieve sideband information.
- void setDefaultReadDelegate (MemoryReadDelegate delegate=MemoryReadDelegate())

  Set default read function for all subsequently added memory spaces.
- void setDefaultTranslateDelegate (MemoryAddressTranslateDelegate delegate=MemoryAddressTranslateDelegate())

  Set the default memory translation delegate.
- void setDefaultWriteDelegate (MemoryWriteDelegate delegate=MemoryWriteDelegate())

  Set default write function for all subsequently added memory spaces.

8.27.1 Detailed Description

Memory add-on for IrisInstance.
This class is used by instances to expose their own memory.
It implements all Iris memory*() functions. It maintains/implements two main things:

- Memory space meta information (exposed by memory_getMemorySpaces()).
- Forwarding memory read/write and address translate accesses to functions with a simple prototype which is easy to implement by components, hiding a lot of the complexity of memory_read(), memory_write(), and memory_translateAddress().

Example usage:

```
irisInstance = new iris::IrisInstance(irisInterface, instanceName);
 irisInstanceMemory = new iris::IrisInstanceMemory(irisInstance);
// Use these delegates for read/write for all following memory spaces.
irisInstanceMemory->setDefaultReadDelegate<DummyComponent, &DummyComponent::readMemory>(this);
irisInstanceMemory->setDefaultWriteDelegate<DummyComponent, &DummyComponent::writeMemory>(this);
irisInstanceMemory->addMemorySpace("Memory"); // Add a memory address space.
```

See setDefaultReadDelegate() for an example of read/write delegates.
See DummyComponent.h for a working example.

See also

IrisInstanceBuilder memory APIs

8.27.2 Constructor & Destructor Documentation

8.27.2.1 IrisInstanceMemory()

```
iris::IrisInstanceMemory::IrisInstanceMemory (
            IrisInstance * irisInstance = 0 )
```

Construct an IrisInstanceMemory.
Optionally attaches to an IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.27.3 Member Function Documentation

8.27.3.1 addAddressTranslation()

```
AddressTranslationInfoAndAccess & iris::IrisInstanceMemory::addAddressTranslation (
            MemorySpaceId inSpaceId,
            MemorySpaceId outSpaceId,
            const std::string & description )
```

Add one memory address translation as well as the translate interface.

Parameters

inSpaceId	Memory space id for the input memory space of this translation.
outSpaceId	Memory space id for the output memory space of this translation.
description	A human-readable description of this translation.

Returns

A reference to an AddressTranslationInfoAndAccess object for the new translation. This reference is valid until the next time addAddressTranslation() is called.

8.27.3.2 addMemorySpace()

```
SpaceInfoAndAccess & iris::IrisInstanceMemory::addMemorySpace (
            const std::string & name )
```

Add meta information for one memory space.

Parameters

name	Name of the memory space.

Returns

A reference to a SpaceInfoAndAccess object for this new memory space. This reference is valid until the next time addMemorySpace() is called.

8.27.3.3 attachTo()

```
void iris::IrisInstanceMemory::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.27.3.4 setDefaultGetSidebandInfoDelegate()

```
void iris::IrisInstanceMemory::setDefaultGetSidebandInfoDelegate (
            MemoryGetSidebandInfoDelegate delegate = MemoryGetSidebandInfoDelegate() )  [inline]
```

Set the default delegate to retrieve sideband information.

Parameters

delegate	Delegate object which will be called to get sideband information for a memory space.

8.27.3.5 setDefaultReadDelegate()

```
void iris::IrisInstanceMemory::setDefaultReadDelegate (
            MemoryReadDelegate delegate = MemoryReadDelegate() )  [inline]
```

Set default read function for all subsequently added memory spaces.

Parameters

delegate	Delegate object which will be called to read memory.

8.27.3.6 setDefaultTranslateDelegate()

```
void iris::IrisInstanceMemory::setDefaultTranslateDelegate (
            MemoryAddressTranslateDelegate delegate = MemoryAddressTranslateDelegate() )
[inline]
```

Set the default memory translation delegate.

Parameters

delegate	Delegate object which will be called to translate addresses.

8.27.3.7 setDefaultWriteDelegate()

```
void iris::IrisInstanceMemory::setDefaultWriteDelegate (
            MemoryWriteDelegate delegate = MemoryWriteDelegate() )  [inline]
```

Set default write function for all subsequently added memory spaces.

Parameters

delegate	Delegate object which will be called to write memory.

The documentation for this class was generated from the following file:

- IrisInstanceMemory.h

8.28 iris::IrisInstancePerInstanceExecution Class Reference

Per-instance execution control add-on for IrisInstance.

`#include <IrisInstancePerInstanceExecution.h>`

Public Member Functions

- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstancePerInstanceExecution add-on to a specific IrisInstance.
- IrisInstancePerInstanceExecution (IrisInstance *irisInstance=nullptr)

  Construct an IrisInstancePerInstanceExecution add-on.
- void setExecutionStateGetDelegate (PerInstanceExecutionStateGetDelegate delegate)

  Set the delegate for getting execution state.
- void setExecutionStateSetDelegate (PerInstanceExecutionStateSetDelegate delegate)

  Set the delegate for setting execution state.

8.28.1 Detailed Description

Per-instance execution control add-on for IrisInstance.
This class is used by instances to support per-instance execution control functionality.
This class implements all Iris perInstanceExecution*() functions.

8.28.2 Constructor & Destructor Documentation

8.28.2.1 IrisInstancePerInstanceExecution()

```
iris::IrisInstancePerInstanceExecution::IrisInstancePerInstanceExecution (
            IrisInstance * irisInstance = nullptr )
```

Construct an IrisInstancePerInstanceExecution add-on.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.28.3 Member Function Documentation

8.28.3.1 attachTo()

```
void iris::IrisInstancePerInstanceExecution::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstancePerInstanceExecution add-on to a specific IrisInstance.
This should only be used if no instance was attached when this object was constructed.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.28.3.2 setExecutionStateGetDelegate()

```
void iris::IrisInstancePerInstanceExecution::setExecutionStateGetDelegate (
            PerInstanceExecutionStateGetDelegate delegate )
```

Set the delegate for getting execution state.

Parameters

delegate	A delegate object which will be called to get the current execution state for the attached instance.

8.28.3.3 setExecutionStateSetDelegate()

```
void iris::IrisInstancePerInstanceExecution::setExecutionStateSetDelegate (
            PerInstanceExecutionStateSetDelegate delegate )
```

Set the delegate for setting execution state.

Parameters

delegate	A delegate object which will be called to set execution state for the attached instance.

The documentation for this class was generated from the following file:

- IrisInstancePerInstanceExecution.h

8.29 iris::IrisInstanceResource Class Reference

Resource add-on for IrisInstance.

`#include <IrisInstanceResource.h>`

Classes

- struct ResourceInfoAndAccess

  Entry in 'resourceInfos'.

Public Member Functions

- ResourceInfoAndAccess & addResource (const std::string &type, const std::string &name, const std::string &description)

  Add a new resource.
- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- void beginResourceGroup (const std::string &name, const std::string &description, uint64_t startSubRscId=IRIS_UINT64_MAX, const std::string &cname=std::string())

  Begin a new resource group.
- ResourceInfoAndAccess * getResourceInfo (ResourceId rscId)

  Get the resource info for a resource that was already added.
- IrisInstanceResource (IrisInstance *irisInstance=0)

  Construct an IrisInstanceResource.
- void setNextSubRscId (ResourceId nextSubRscId_)

  Set next subRscId.
- void setTag (ResourceId rscId, const std::string &tag)

  Set a tag for a specific resource.

Static Public Member Functions

- static void calcHierarchicalNames (std::vector< ResourceInfo > &resourceInfos, const std::vector< ResourceGroupInfo > &groupInfos=std::vector< ResourceGroupInfo >())

  Calculate hierarchicalName and hierarchicalCName for all RegisterInfos.
- static void makeNamesHierarchical (std::vector< ResourceInfo > &resourceInfos, const std::vector< ResourceGroupInfo > &groupInfos=std::vector< ResourceGroupInfo >())

  Make name and cname of RegisterInfos hierarchical.

Protected Member Functions

- void **impl_resource_getList** (IrisReceivedRequest &request)
- void **impl_resource_getListOfResourceGroups** (IrisReceivedRequest &request)
- void **impl_resource_getResourceInfo** (IrisReceivedRequest &request)
- void **impl_resource_read** (IrisReceivedRequest &request)
- void **impl_resource_write** (IrisReceivedRequest &request)

8.29.1 Detailed Description

Resource add-on for IrisInstance.
This class implements all Iris resource∗() functions. It maintains/implements two main things:

- Resource meta information that is exposed by resource_getList() and resource_getListOfResourceGroups().
- Forwarding resource read/write accesses to functions with a simple prototype which is easy to implement by components, hiding a lot of the complexity of resource_read() and resource_write().

In most cases, an instance should not use IrisInstanceResource directly but should use IrisInstanceBuilder instead.

8.29.2 Constructor & Destructor Documentation

8.29.2.1 IrisInstanceResource()

```
iris::IrisInstanceResource::IrisInstanceResource (
            IrisInstance * irisInstance = 0 )
```

Construct an IrisInstanceResource.
Optionally attaches to an IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.29.3 Member Function Documentation

8.29.3.1 addResource()

```
ResourceInfoAndAccess & iris::IrisInstanceResource::addResource (
            const std::string & type,
            const std::string & name,
            const std::string & description )
```

Add a new resource.

Parameters

type	The type of the resource. This should be one of: • "numeric" • "numericFp" • "String" • "noValue"
name	The name of the resource.

Parameters

description	A human-readable description of the resource.

Returns

A reference to a ResourceInfoAndAccess object for this new resource. This reference is valid until the next time addResource() is called.

8.29.3.2 attachTo()

```
void iris::IrisInstanceResource::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.29.3.3 beginResourceGroup()

```
void iris::IrisInstanceResource::beginResourceGroup (
            const std::string & name,
            const std::string & description,
            uint64_t startSubRscId = IRIS_UINT64_MAX,
            const std::string & cname = std::string() )
```

Begin a new resource group.
This method has these effects:

- Add a resource group (only if it does not yet exist).
- Assign all resources that are added through addResource() calls to this group.

Parameters

name	The name of the resource group.
description	A description of this resource group.
startSubRscId	If not IRIS_UINT64_MAX start counting from this subRscId when new resources are added.
cname	A C identifier version of the resource name if different from *name*.

8.29.3.4 calcHierarchicalNames()

```
static void iris::IrisInstanceResource::calcHierarchicalNames (
            std::vector< ResourceInfo > & resourceInfos,
            const std::vector< ResourceGroupInfo > & groupInfos = std::vector< ResourceGroupInfo >()
) [static]
```

Calculate hierarchicalName and hierarchicalCName for all RegisterInfos.
RegisterInfo.hierarchicalName and RegisterInfo.hierarchicalCName are set to the hierarchical name for each resource such that a child register X of parent FLAGS gets hierarchicalName=FLAGS.X and hierarchicalCName=FLAGS_X, similarly also for deeper nesting levels.

This functionality is not an Iris interface but just a convenience function for simple clients. The ResourceInfos returned by IrisInstance::getResourceInfo*() have already hierarchical names.
No errors are generated for missing parent resources. parentRscId links to missing parent resources are silently ignored. The intended usage is to call this function on a list containing all resources or all registers of an instance, so that all parent links can be resolved.

Parameters

resourceInfos	Array of all ResourceInfos of an instance.

8.29.3.5 getResourceInfo()

```
ResourceInfoAndAccess * iris::IrisInstanceResource::getResourceInfo (
            ResourceId rscId )
```

Get the resource info for a resource that was already added.

Parameters

rsc↩ Id	A resource id for a resource that was already added.

Returns

A pointer to the ResourceInfoAndAccess object for the requested resource. This pointer is valid until the next call to addResource(). If *rscId* is not a valid id, this function returns nullptr.

8.29.3.6 makeNamesHierarchical()

```
static void iris::IrisInstanceResource::makeNamesHierarchical (
            std::vector< ResourceInfo > & resourceInfos,
            const std::vector< ResourceGroupInfo > & groupInfos = std::vector< ResourceGroupInfo >()
) [static]
```

Make name and cname of RegisterInfos hierarchical.
Legacy function overwriting ResourceInfo.name/cname.
This function calculates the hierarchical names using calcHierarchicalNames() and then copies ResourceInfo.↩ hierarchicalName/hierarchicalCName into ResourceInfo.name/cname info, respectively.
Consider using calcHierarchicalNames() which does not alter the original resource information.

Parameters

resourceInfos	Array of all ResourceInfos of an instance.

8.29.3.7 setNextSubRscId()

```
void iris::IrisInstanceResource::setNextSubRscId (
            ResourceId nextSubRscId_ ) [inline]
```

Set next subRscId.
Resources that are added following this call are assigned subRscIds starting at nextSubRscId unless nextSubRscId is IRIS_UINT64_MAX, in which case all further resources are assigned IRIS_UINT64_MAX as the subRscId

Parameters

*nextSubRsc↩ Id_*	Next subRscId

8.29.3.8 setTag()

```
void iris::IrisInstanceResource::setTag (
            ResourceId rscId,
            const std::string & tag )
```

Set a tag for a specific resource.

Parameters

rsc↩ Id	Resource Id for the resource that will have this tag set.
tag	Name of the boolean tag which will be set to true.

See also

IrisInstanceBuilder::setTag

The documentation for this class was generated from the following file:

- IrisInstanceResource.h

8.30 iris::IrisInstanceSemihosting Class Reference

Public Member Functions

- void attachTo (IrisInstance *iris_instance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- void **enableExtensions** ()

  Instances that support semihosting extensions should call this method to enable the `IRIS_SEMIHOSTING_↩ CALL_EXTENSION` event.
- **IrisInstanceSemihosting** (IrisInstance *iris_instance=nullptr, IrisInstanceEvent *inst_event=nullptr)
- std::vector< uint8_t > readData (uint64_t fDes, uint64_t max_size=0, uint64_t flags=semihost::DEFAULT)

  Read data for a given file descriptor.
- std::pair< bool, uint64_t > semihostedCall (uint64_t operation, uint64_t parameter)

  Allow a client to perform a semihosting extension defined by operation and parameter.
- void setEventHandler (IrisInstanceEvent *handler)

  Set the corresponding IrisInstanceEvent object to use to manage semihosting events.
- void **unblock** ()

  Request premature exit from any blocking requests that are currently blocked.
- bool **writeData** (uint64_t fDes, const uint8_t *data, uint64_t size)

8.30.1 Member Function Documentation

8.30.1.1 attachTo()

```
void iris::IrisInstanceSemihosting::attachTo (
            IrisInstance * iris_instance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

iris_instance	The instance to attach to.

8.30.1.2 readData()

```
std::vector< uint8_t > iris::IrisInstanceSemihosting::readData (
            uint64_t fDes,
            uint64_t max_size = 0,
            uint64_t flags = semihost::DEFAULT )
```

Read data for a given file descriptor.
The exact behavior of this method depends on the value of the max_size and flags parameters. If the NONBLOCK flag is set, the method returns immediately with whatever data is already buffered, if any. If NONBLOCK is not set, the method blocks until data is available. Iris messages continue to be processed while this methods blocks. If max_size is not zero, then at most max_size bytes will be returned.

Parameters

fDes	File descriptor to read from. Usually semihost::STDIN.
max_size	The maximum amount of bytes to read or zero for no limit.
flags	A bitwise OR of Semihosting data request flag constants

Returns

A vector of data that was read.

8.30.1.3 semihostedCall()

```
std::pair< bool, uint64_t > iris::IrisInstanceSemihosting::semihostedCall (
            uint64_t operation,
            uint64_t parameter )
```

Allow a client to perform a semihosting extension defined by *operation* and *parameter*.
This might implement a user-defined operation or override the default implementation for a predefined operation.

Parameters

operation	A number indicating the operation to perform. This is defined by the semihosting standard for standard operations or by the client for user-defined operations.
parameter	A parameter to the operation. This meaning of this parameter is defined by the operation.

Returns

A pair of (bool success, uint64_t result). If status is true, a client performed the function and returned the value in result. If status is false, no client performed the function and result is 0.

8.30.1.4 setEventHandler()

```
void iris::IrisInstanceSemihosting::setEventHandler (
            IrisInstanceEvent * handler )
```

Set the corresponding IrisInstanceEvent object to use to manage semihosting events.
This must not be called more than once and must be called with an Event add-on that is attached to the same IrisInstance as this semihosting add-on.

Parameters

handler	The event add-on for this Iris instance.

The documentation for this class was generated from the following file:

- IrisInstanceSemihosting.h

8.31 iris::IrisInstanceSimulation Class Reference

An IrisInstance add-on that adds simulation functions for the SimulationEngine instance.
`#include <IrisInstanceSimulation.h>`

Public Member Functions

- void attachTo (IrisInstance *iris_instance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- void enterPostInstantiationPhase ()

  Move from the pre-instantiation to the post-instantiation phase.
- IrisInstanceSimulation (IrisInstance *iris_instance=nullptr, IrisConnectionInterface *connection_interface=nullptr)

  Construct an IrisInstanceSimulation add-on.
- void notifySimPhase (uint64_t time, IrisSimulationPhase phase, const IrisValueMap *fields=nullptr)

  Emit an IRIS_SIM_PHASE event for the supplied phase.*
- void registerSimEventsOnGlobalInstance ()

  Register all simulation engine events as proxy events on the global iris instance.
- void setConnectionInterface (IrisConnectionInterface *connection_interface_)

  Set the IrisConnectionInterface to use for the instantiation.
- void setEventHandler (IrisInstanceEvent *handler)

  Set up IRIS_SIM_PHASE events.*
- template<IrisErrorCode(*)(std::vector< ResourceInfo > &) FUNC>
  void setGetParameterInfoDelegate (bool cache_result=true)

  Set the getParameterInfo() delegate.
- void setGetParameterInfoDelegate (SimulationGetParameterInfoDelegate delegate, bool cache_result=true)

  Set the getParameterInfo() delegate.
- template<typename T , IrisErrorCode(T::*)(std::vector< ResourceInfo > &) METHOD>
  void setGetParameterInfoDelegate (T *instance, bool cache_result=true)

  Set the getParameterInfo() delegate.
- template<IrisErrorCode(*)(InstantiationResult &) FUNC>
  void setInstantiateDelegate ()

  Set the instantiate() delegate.
- void setInstantiateDelegate (SimulationInstantiateDelegate delegate)

  Set the instantiate() delegate.
- template<typename T , IrisErrorCode(T::*)(InstantiationResult &) METHOD>
  void setInstantiateDelegate (T *instance)

  Set the instantiate() delegate.
- void setLogLevel (unsigned logLevel_)

  Set log level (0-1).
- template<IrisErrorCode(*)() FUNC>
  void setRequestShutdownDelegate ()

  Set the requestShutdown() delegate.
- void setRequestShutdownDelegate (SimulationRequestShutdownDelegate delegate)

  Set the requestShutdown() delegate.

- template<typename T , IrisErrorCode(T::*)() METHOD>
  void setRequestShutdownDelegate (T *instance)

  Set the requestShutdown() delegate.
- template<IrisErrorCode(*)(const IrisSimulationResetContext &) FUNC>
  void setResetDelegate ()

  Set the reset() delegate.
- void setResetDelegate (SimulationResetDelegate delegate)

  Set the reset() delegate.
- template<typename T , IrisErrorCode(T::*)(const IrisSimulationResetContext &) METHOD>
  void setResetDelegate (T *instance)

  Set the reset() delegate.
- template<IrisErrorCode(*)(const InstantiationParameterValue &) FUNC>
  void setSetParameterValueDelegate ()

  Set the setParameterValue() delegate.
- void setSetParameterValueDelegate (SimulationSetParameterValueDelegate delegate)

  Set the setParameterValue() delegate.
- template<typename T , IrisErrorCode(T::*)(const InstantiationParameterValue &) METHOD>
  void setSetParameterValueDelegate (T *instance)

  Set the setParameterValue() delegate.

Static Public Member Functions

- static std::string getSimulationPhaseDescription (IrisSimulationPhase phase)

  Get dexcription string for a simulation phase.
- static std::string getSimulationPhaseName (IrisSimulationPhase phase)

  Get name of the enum symbol for name.

8.31.1 Detailed Description

An IrisInstance add-on that adds simulation functions for the SimulationEngine instance.

8.31.2 Constructor & Destructor Documentation

8.31.2.1 IrisInstanceSimulation()

```
iris::IrisInstanceSimulation::IrisInstanceSimulation (
            IrisInstance * iris_instance = nullptr,
            IrisConnectionInterface * connection_interface = nullptr )
```

Construct an IrisInstanceSimulation add-on.

Parameters

iris_instance	The IrisInstance to attach this add-on to.
connection_interface	The connection interface that will be used when the simulation is instantiated.

8.31.3 Member Function Documentation

8.31.3.1 attachTo()

```
void iris::IrisInstanceSimulation::attachTo (
            IrisInstance * iris_instance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

iris_instance	The IrisInstance to attach to.

8.31.3.2 enterPostInstantiationPhase()

```
void iris::IrisInstanceSimulation::enterPostInstantiationPhase ( )
```

Move from the pre-instantiation to the post-instantiation phase.
This effects which functions are published. Only call this function if the simulation is instantiated outside of Iris. This object automatically enters post-instantiation phase when the simulation is successfully instantiated by an Iris call to simulation_instantiate().

8.31.3.3 getSimulationPhaseDescription()

```
static std::string iris::IrisInstanceSimulation::getSimulationPhaseDescription (
            IrisSimulationPhase phase )  [static]
```

Get dexcription string for a simulation phase.
This is a free form single line text ending with a dot.

8.31.3.4 getSimulationPhaseName()

```
static std::string iris::IrisInstanceSimulation::getSimulationPhaseName (
            IrisSimulationPhase phase )  [static]
```

Get name of the enum symbol for name.
Example: getSimulationPhaseName(IRIS_SIM_PHASE_INIT) returns "IRIS_SIM_PHASE_INIT".

8.31.3.5 notifySimPhase()

```
void iris::IrisInstanceSimulation::notifySimPhase (
            uint64_t time,
            IrisSimulationPhase phase,
            const IrisValueMap * fields = nullptr )
```

Emit an IRIS_SIM_PHASE∗ event for the supplied phase.

Parameters

time	The simulation time at which the event occurred.
phase	The simulation phase that was reached.
fields	Sim-phase events usually do not have fields.

8.31.3.6 registerSimEventsOnGlobalInstance()

```
void iris::IrisInstanceSimulation::registerSimEventsOnGlobalInstance ( )
```

Register all simulation engine events as proxy events on the global iris instance.
This function should be called after an iris instance has been attached to IrisInstanceSimulation object (IrisInstanceSimulation::attachTo). This will ensure that the simulation engine iris instance i.e. iris_instance is available to call the register API. This function should be called after event handler has been set for IrisInstanceSimulation object (IrisInstanceSimulation::setEventHandler). This will ensure that all simulation engine events are available in simulation engine event handler. This function should be called after an IrisIntanceEvent has been attached to iris_instance (IrisInstanceEvent::attachTo). This will ensure that event functions have been registered on simulation engine iris instance.

8.31.3.7 setConnectionInterface()

```
void iris::IrisInstanceSimulation::setConnectionInterface (
            IrisConnectionInterface * connection_interface_ )  [inline]
```

Set the IrisConnectionInterface to use for the instantiation.
This will be passed to the instantiate() delegate when the simulation is instantiated.

8.31.3.8 setEventHandler()

```
void iris::IrisInstanceSimulation::setEventHandler (
            IrisInstanceEvent * handler )
```

Set up IRIS_SIM_PHASE∗ events.

Parameters

handler	An IrisInstanceEvent add-on that is attached to the same instance as this add-on.

8.31.3.9 setGetParameterInfoDelegate() [1/3]

```
template<IrisErrorCode(*)(std::vector< ResourceInfo > &) FUNC>
void iris::IrisInstanceSimulation::setGetParameterInfoDelegate (
            bool cache_result = true )  [inline]
```

Set the getParameterInfo() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is a getParameterInfo delegate.

Parameters

cache_result	If true, the delegate is only called once and the result is cached and used for subsequent calls to `simulation_getInstantiationParameterInfo()`. If false, the result is not cached and the delegate is called every time.

8.31.3.10 setGetParameterInfoDelegate() [2/3]

```
void iris::IrisInstanceSimulation::setGetParameterInfoDelegate (
            SimulationGetParameterInfoDelegate delegate,
            bool cache_result = true )  [inline]
```

Set the getParameterInfo() delegate.

Parameters

delegate	A delegate object that is called to get instantiation parameter information for the simulation.
cache_result	If true, the delegate is only called once and the result is cached and used for subsequent calls to `simulation_getInstantiationParameterInfo()`. If false, the result is not cached and the delegate is called every time.

8.31.3.11 setGetParameterInfoDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(std::vector< ResourceInfo > &) METHOD>
void iris::IrisInstanceSimulation::setGetParameterInfoDelegate (
            T * instance,
            bool cache_result = true )  [inline]
```

Set the getParameterInfo() delegate.
Set the delegate to call a method in class T.

Template Parameters

T	Class that defines a getParameterInfo delegate method.
METHOD	A method of class *T* that is a getParameterInfo delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.
cache_result	If true, the delegate is called once and the result is cached and used for subsequent calls to `simulation_getInstantiationParameterInfo()`. If false, the result is not cached and the delegate is called every time.

8.31.3.12 setInstantiateDelegate() [1/3]

```
template<IrisErrorCode(*)(InstantiationResult &) FUNC>
void iris::IrisInstanceSimulation::setInstantiateDelegate ( )  [inline]
```

Set the instantiate() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is an instantiate delegate.

8.31.3.13 setInstantiateDelegate() [2/3]

```
void iris::IrisInstanceSimulation::setInstantiateDelegate (
            SimulationInstantiateDelegate delegate )  [inline]
```

Set the instantiate() delegate.

Parameters

delegate	A delegate object that will be called to instantiate the simulation.

8.31.3.14 setInstantiateDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(InstantiationResult &) METHOD>
void iris::IrisInstanceSimulation::setInstantiateDelegate (
            T * instance )  [inline]
```

Set the instantiate() delegate.
Set the delegate to call a method in class T.

Template Parameters

T	Class that defines an instantiate delegate method.
METHOD	A method of class *T* that is an instantiate delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.

8.31.3.15 setLogLevel()

```
void iris::IrisInstanceSimulation::setLogLevel (
            unsigned logLevel_ )
```

Set log level (0-1).
Set log level (0-1).

8.31.3.16 setRequestShutdownDelegate() [1/3]

```
template<IrisErrorCode(*)() FUNC>
void iris::IrisInstanceSimulation::setRequestShutdownDelegate ( )  [inline]
```

Set the requestShutdown() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is a requestShutdown delegate.

8.31.3.17 setRequestShutdownDelegate() [2/3]

```
void iris::IrisInstanceSimulation::setRequestShutdownDelegate (
            SimulationRequestShutdownDelegate delegate )  [inline]
```

Set the requestShutdown() delegate.

Parameters

delegate	A delegate object that will be called to request that the simulation be shut down.

8.31.3.18 setRequestShutdownDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)() METHOD>
void iris::IrisInstanceSimulation::setRequestShutdownDelegate (
            T * instance )  [inline]
```

Set the requestShutdown() delegate.
Set the delegate to call a method in class T.

Template Parameters

T	Class that defines a requestShutdown delegate method.
METHOD	A method of class *T* that is a requestShutdown delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.

8.31.3.19 setResetDelegate() [1/3]

```
template<IrisErrorCode(*)(const IrisSimulationResetContext &) FUNC>
void iris::IrisInstanceSimulation::setResetDelegate ( )  [inline]
```

Set the reset() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is a reset delegate.

8.31.3.20 setResetDelegate() [2/3]

```
void iris::IrisInstanceSimulation::setResetDelegate (
            SimulationResetDelegate delegate )  [inline]
```

Set the reset() delegate.

Parameters

delegate	A delegate object which will be called to reset the simulation.

8.31.3.21 setResetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const IrisSimulationResetContext &) METHOD>
void iris::IrisInstanceSimulation::setResetDelegate (
            T * instance )  [inline]
```

Set the reset() delegate.
Set the delegate to call a method in class T.

Template Parameters

T	Class that defines a reset delegate method.
METHOD	A method of class *T* that is a reset delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.

8.31.3.22 setSetParameterValueDelegate() [1/3]

```
template<IrisErrorCode(*)(const InstantiationParameterValue &) FUNC>
void iris::IrisInstanceSimulation::setSetParameterValueDelegate ( )  [inline]
```

Set the setParameterValue() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is a setParameterValue delegate.

8.31.3.23 setSetParameterValueDelegate() [2/3]

```
void iris::IrisInstanceSimulation::setSetParameterValueDelegate (
            SimulationSetParameterValueDelegate delegate ) [inline]
```

Set the setParameterValue() delegate.

Parameters

delegate	A delegate object that is called to set instantiation parameter values before instantiation.

8.31.3.24 setSetParameterValueDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const InstantiationParameterValue &) METHOD>
void iris::IrisInstanceSimulation::setSetParameterValueDelegate (
            T * instance ) [inline]
```

Set the setParameterValue() delegate.
Set the delegate to call a method in class T.

Template Parameters

T	Class that defines a setParameterValue delegate method.
METHOD	A method of class *T* that is a setParameterValue delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.

The documentation for this class was generated from the following file:

- IrisInstanceSimulation.h

8.32 iris::IrisInstanceSimulationTime Class Reference

Simulation time add-on for IrisInstance.
`#include <IrisInstanceSimulationTime.h>`

Public Member Functions

- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstance add-on to a specific IrisInstance.
- IrisInstanceSimulationTime (IrisInstance *iris_instance=nullptr, IrisInstanceEvent *inst_event=nullptr)

  Construct an IrisInstanceSimulationTime add-on.
- void **notifySimulationTimeEvent** (uint64_t reason=TIME_EVENT_UNKNOWN)

  Generate the IRIS_SIMULATION_TIME_EVENT event callback.
- void registerSimTimeEventsOnGlobalInstance ()

  Register all simulation time events as proxy events on the global iris instance.

- void setEventHandler (IrisInstanceEvent *handler)

  Set the event handler to use to send simulation time-related events.
- template<IrisErrorCode(*)(uint64_t &, uint64_t &, bool &) FUNC>
  void setSimTimeGetDelegate ()

  Set the getTime() delegate.
- void setSimTimeGetDelegate (SimulationTimeGetDelegate delegate)

  Set the getTime() delegate.
- template<typename T , IrisErrorCode(T::*)(uint64_t &, uint64_t &, bool &) METHOD>
  void setSimTimeGetDelegate (T *instance)

  Set the getTime() delegate.
- void setSimTimeNotifyStateChanged (std::function< void()> func)

  Set the notifyStateChanged() delegate.
- template<IrisErrorCode(*)() FUNC>
  void setSimTimeRunDelegate ()

  Set the run() delegate.
- void setSimTimeRunDelegate (SimulationTimeRunDelegate delegate)

  Set the run() delegate.
- template<typename T , IrisErrorCode(T::*)() METHOD>
  void setSimTimeRunDelegate (T *instance)

  Set the run() delegate.
- template<IrisErrorCode(*)() FUNC>
  void setSimTimeStopDelegate ()

  Set the stop() delegate.
- void setSimTimeStopDelegate (SimulationTimeStopDelegate delegate)

  Set the stop() delegate.
- template<typename T , IrisErrorCode(T::*)() METHOD>
  void setSimTimeStopDelegate (T *instance)

  Set the stop() delegate.

8.32.1 Detailed Description

Simulation time add-on for IrisInstance.

8.32.2 Constructor & Destructor Documentation

8.32.2.1 IrisInstanceSimulationTime()

```
iris::IrisInstanceSimulationTime::IrisInstanceSimulationTime (
            IrisInstance * iris_instance = nullptr,
            IrisInstanceEvent * inst_event = nullptr )
```

Construct an IrisInstanceSimulationTime add-on.

Parameters

iris_instance	An IrisInstance to attach this add-on to.
inst_event	An IrisInstanceEvent add-on that is already attached to IrisInstance. This is used to set up simulation time events.

8.32.3 Member Function Documentation

8.32.3.1 attachTo()

```
void iris::IrisInstanceSimulationTime::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	An IrisInstance to attach this add-on to.

8.32.3.2 registerSimTimeEventsOnGlobalInstance()

```
void iris::IrisInstanceSimulationTime::registerSimTimeEventsOnGlobalInstance ( )
```

Register all simulation time events as proxy events on the global iris instance.
This function should be called after an iris instance has been attached to IrisInstanceSimulationTime object (IrisInstanceSimulationTime::attachTo). This will ensure that the simulation time iris instance i.e. iris_↩ instance is available to call the register API. This function should be called after event handler has been set for IrisInstanceSimulationTime object (IrisInstanceSimulationTime::setEventHandler). This will ensure that all simulation time events are available in simulation time event handler. This function should be called after an IrisIntance↩ Event has been attached to iris_instance (IrisInstanceEvent::attachTo). This will ensure that event functions have been registered on simulation time iris instance.

8.32.3.3 setEventHandler()

```
void iris::IrisInstanceSimulationTime::setEventHandler (
            IrisInstanceEvent * handler )
```

Set the event handler to use to send simulation time-related events.

Parameters

handler	An IrisInstanceEvent add-on that is already attached to IrisInstance. This is used to set up simulation time events.

8.32.3.4 setSimTimeGetDelegate() [1/3]

```
template<IrisErrorCode(*)(uint64_t &, uint64_t &, bool &) FUNC>
void iris::IrisInstanceSimulationTime::setSimTimeGetDelegate ( )  [inline]
```

Set the getTime() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is a getTime delegate.

8.32.3.5 setSimTimeGetDelegate() [2/3]

```
void iris::IrisInstanceSimulationTime::setSimTimeGetDelegate (
            SimulationTimeGetDelegate delegate )  [inline]
```

Set the getTime() delegate.

Parameters

delegate	A delegate that is called to get the current simulation time.

8.32.3.6 setSimTimeGetDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(uint64_t &, uint64_t &, bool &) METHOD>
void iris::IrisInstanceSimulationTime::setSimTimeGetDelegate (
            T * instance )  [inline]
```

Set the getTime() delegate.

Template Parameters

T	Class that defines a getTime delegate method.
METHOD	A method of class *T* that is a getTime delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.

8.32.3.7 setSimTimeNotifyStateChanged()

```
void iris::IrisInstanceSimulationTime::setSimTimeNotifyStateChanged (
            std::function< void()> func )  [inline]
```

Set the notifyStateChanged() delegate.
The semantics of this delegate is to emit a IRIS_SIMULATION_TIME_EVENT(REASON=STATE_CHANGED) event, usually by calling notifySimulationTimeEvent(TIME_EVENT_STATE_CHANGED). Ideally this is done with a small delay so that multiple successive calls to simulationTime_notifyStateChanged() cause only one IRIS_↩SIMULATION_TIME_EVENT(REASON=STATE_CHANGED) event. In other words multiple calls to simulation↩Time_notifyStateChanged() should be aggregated into one IRIS_SIMULATION_TIME_EVENT(REASON=STATE↩_CHANGED) event. The delay from the first call to simulationTime_notifyStateChanged() to the IRIS_↩SIMULATION_TIME_EVENT(REASON=STATE_CHANGED) event should be approximately 500 ms.
The default implementation of this delegate immediately emits a IRIS_SIMULATION_TIME_EVENT(REASON=STATE↩_CHANGED) event and does not aggregate multiple calls to simulationTime_notifyStateChanged().

Parameters

func	A function which calls notifySimulationTimeEvent() within the next 500 ms.

8.32.3.8 setSimTimeRunDelegate() [1/3]

```
template<IrisErrorCode(*)() FUNC>
void iris::IrisInstanceSimulationTime::setSimTimeRunDelegate ( )  [inline]
```

Set the run() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is a run delegate.

8.32.3.9 setSimTimeRunDelegate() [2/3]

```
void iris::IrisInstanceSimulationTime::setSimTimeRunDelegate (
```

SimulationTimeRunDelegate *delegate*) [inline]

Set the run() delegate.

Parameters

delegate	A delegate that is called to start/resume progress of simulation time.

8.32.3.10 setSimTimeRunDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)() METHOD>
void iris::IrisInstanceSimulationTime::setSimTimeRunDelegate (
            T * instance ) [inline]
```

Set the run() delegate.

Template Parameters

T	Class that defines a run delegate method.
METHOD	A method of class *T* that is a run delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.

8.32.3.11 setSimTimeStopDelegate() [1/3]

```
template<IrisErrorCode(*)() FUNC>
void iris::IrisInstanceSimulationTime::setSimTimeStopDelegate ( ) [inline]
```

Set the stop() delegate.
Set the delegate to a global function.

Template Parameters

FUNC	A function that is a stop delegate.

8.32.3.12 setSimTimeStopDelegate() [2/3]

```
void iris::IrisInstanceSimulationTime::setSimTimeStopDelegate (
            SimulationTimeStopDelegate delegate ) [inline]
```

Set the stop() delegate.

Parameters

delegate	A delegate that is called to stop the progress of simulation time.

8.32.3.13 setSimTimeStopDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)() METHOD>
void iris::IrisInstanceSimulationTime::setSimTimeStopDelegate (
```

```
            T * instance )  [inline]
```

Set the stop() delegate.

Template Parameters

T	Class that defines a stop delegate method.
METHOD	A method of class *T* that is a stop delegate.

Parameters

instance	An instance of class *T* on which *METHOD* should be called.

The documentation for this class was generated from the following file:

- IrisInstanceSimulationTime.h

8.33 iris::IrisInstanceStep Class Reference

Step add-on for IrisInstance.

`#include <IrisInstanceStep.h>`

Public Member Functions

- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstanceStep add-on to a specific IrisInstance.
- IrisInstanceStep (IrisInstance *irisInstance=nullptr)

  Construct an IrisInstanceStep add-on.
- void setRemainingStepGetDelegate (RemainingStepGetDelegate delegate)

  Set the delegate for getting the remaining steps.
- void setRemainingStepSetDelegate (RemainingStepSetDelegate delegate)

  Set the delegate for setting the remaining steps.
- void setStepCountGetDelegate (StepCountGetDelegate delegate)

  Set the delegate for getting the step count.

8.33.1 Detailed Description

Step add-on for IrisInstance.
This is used by instances to support stepping functionality.
This class implements all Iris step*() functions.

8.33.2 Constructor & Destructor Documentation

8.33.2.1 IrisInstanceStep()

```
iris::IrisInstanceStep::IrisInstanceStep (
            IrisInstance * irisInstance = nullptr )
```

Construct an IrisInstanceStep add-on.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.33.3 Member Function Documentation

8.33.3.1 attachTo()

```
void iris::IrisInstanceStep::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstanceStep add-on to a specific IrisInstance.
This should only be used if no instance was attached when this object was constructed.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.33.3.2 setRemainingStepGetDelegate()

```
void iris::IrisInstanceStep::setRemainingStepGetDelegate (
            RemainingStepGetDelegate delegate )
```

Set the delegate for getting the remaining steps.

Parameters

delegate	A delegate object that is called to get the remaining steps for the attached instance.

8.33.3.3 setRemainingStepSetDelegate()

```
void iris::IrisInstanceStep::setRemainingStepSetDelegate (
            RemainingStepSetDelegate delegate )
```

Set the delegate for setting the remaining steps.

Parameters

delegate	A delegate object that is called to set the remaining steps for the attached instance.

8.33.3.4 setStepCountGetDelegate()

```
void iris::IrisInstanceStep::setStepCountGetDelegate (
            StepCountGetDelegate delegate )
```

Set the delegate for getting the step count.

Parameters

delegate	A delegate object that is called to get the step count for the attached instance.

The documentation for this class was generated from the following file:

- IrisInstanceStep.h

8.34 iris::IrisInstanceTable Class Reference

Table add-on for IrisInstance.

`#include <IrisInstanceTable.h>`

Classes

- struct TableInfoAndAccess

  Entry in 'tableInfos'.

Public Member Functions

- TableInfoAndAccess & addTableInfo (const std::string &name)

  Add metadata for one table.
- void attachTo (IrisInstance *irisInstance)

  Attach this IrisInstanceTable add-on to a specific IrisInstance.
- IrisInstanceTable (IrisInstance *irisInstance=nullptr)

  Construct an IrisInstanceTable add-on.
- void setDefaultReadDelegate (TableReadDelegate delegate=TableReadDelegate())

  Set the default delegate for reading table data.
- void setDefaultWriteDelegate (TableWriteDelegate delegate=TableWriteDelegate())

  Set the default delegate for writing table data.

8.34.1 Detailed Description

Table add-on for IrisInstance.
This is used by instances to support table functionality.

8.34.2 Constructor & Destructor Documentation

8.34.2.1 IrisInstanceTable()

```
iris::IrisInstanceTable::IrisInstanceTable (
            IrisInstance * irisInstance = nullptr )
```

Construct an IrisInstanceTable add-on.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.34.3 Member Function Documentation

8.34.3.1 addTableInfo()

```
TableInfoAndAccess & iris::IrisInstanceTable::addTableInfo (
            const std::string & name )
```

Add metadata for one table.

Parameters

name	The name of this table.

Returns

A reference to a TableInfoAndAccess object that can be used to set metadata and access delegates for this table.

8.34.3.2 attachTo()

```
void iris::IrisInstanceTable::attachTo (
            IrisInstance * irisInstance )
```

Attach this IrisInstanceTable add-on to a specific IrisInstance.
This should only be used if no instance was attached when this object was constructed.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.34.3.3 setDefaultReadDelegate()

```
void iris::IrisInstanceTable::setDefaultReadDelegate (
            TableReadDelegate delegate = TableReadDelegate() )  [inline]
```

Set the default delegate for reading table data.

Parameters

delegate	A delegate object that is called to read table data for tables in the attached instance that did not set a table-specific delegate.

8.34.3.4 setDefaultWriteDelegate()

```
void iris::IrisInstanceTable::setDefaultWriteDelegate (
            TableWriteDelegate delegate = TableWriteDelegate() )  [inline]
```

Set the default delegate for writing table data.

Parameters

delegate	A delegate object that is called to write table data for tables in the attached instance that did not set a table-specific delegate.

The documentation for this class was generated from the following file:

- IrisInstanceTable.h

8.35 iris::IrisInstantiationContext Class Reference

Provides context when instantiating an Iris instance from a factory.
`#include <IrisInstantiationContext.h>`

Public Member Functions

- void void void error (const std::string &code, const char *format,...) INTERNAL_IRIS_PRINTF(3

  Add an error to the InstantiationResult.
- bool getBoolParameter (const std::string &name)

Get the value of an instantiation parameter as boolean.

- IrisConnectionInterface * getConnectionInterface () const

  Get the connection interface to use to register the instance being instantiated.
- std::string getInstanceName () const

  Get the instance name to use when registering the instance being instantiated.
- const IrisValue & getParameter (const std::string &name)

  Get the value of an instantiation parameter as IrisValue.
- void getParameter (const std::string &name, std::vector< uint64_t > &value)

  Get the value of a large numeric instantiation parameter.
- template<typename T >
  void getParameter (const std::string &name, T &value)

  Get the value of an instantiation parameter.
- uint64_t getRecommendedInstanceFlags () const

  Get the flags to use when registering the instance being instantiated.
- int64_t getS64Parameter (const std::string &name)

  Get the value of an instantiation parameter as int64_t.
- std::string getStringParameter (const std::string &name)

  Get the value of an instantiation parameter as string.
- IrisInstantiationContext * getSubcomponentContext (const std::string &child_name)

  Get an IrisInstanceContext pointer for a subcomponent instance.
- uint64_t getU64Parameter (const std::string &name)

  Get the value of an instantiation parameter as uint64_t.
- **IrisInstantiationContext** (IrisConnectionInterface *connection_interface_, InstantiationResult &result_↩ , const std::vector< ResourceInfo > &param_info_, const std::vector< InstantiationParameterValue > &param_values_, const std::string &prefix_, const std::string &component_name_, uint64_t instance_flags↩ _)
- void void void void parameterError (const std::string &code, const std::string &parameterName, const char *format,...) INTERNAL_IRIS_PRINTF(4

  Add an error to the InstantiationResult.
- void void parameterWarning (const std::string &code, const std::string &parameterName, const char *format,...) INTERNAL_IRIS_PRINTF(4

  Add a warning to the InstantiationResult.
- void warning (const std::string &code, const char *format,...) INTERNAL_IRIS_PRINTF(3

  Add a warning to the InstantiationResult.

8.35.1 Detailed Description

Provides context when instantiating an Iris instance from a factory.

8.35.2 Member Function Documentation

8.35.2.1 error()

```
void void void iris::IrisInstantiationContext::error (
            const std::string & code,
            const char * format,
             ...  )
```

Add an error to the InstantiationResult.

See also

parameterError

Parameters

code	An error code symbol. This should be one of the codes specified for the InstantiationError object.
format	A printf-style format string.
...	Printf substitution arguments.

8.35.2.2 getBoolParameter()

```
bool iris::IrisInstantiationContext::getBoolParameter (
            const std::string & name )  [inline]
```

Get the value of an instantiation parameter as boolean.

Parameters

name	The name of the parameter.

Returns

Boolean value.

8.35.2.3 getConnectionInterface()

```
IrisConnectionInterface * iris::IrisInstantiationContext::getConnectionInterface ( ) const
[inline]
```

Get the connection interface to use to register the instance being instantiated.

Returns

A value to use for the `connection_interface` argument of IrisInstance::IrisInstance().

8.35.2.4 getInstanceName()

```
std::string iris::IrisInstantiationContext::getInstanceName ( ) const  [inline]
```

Get the instance name to use when registering the instance being instantiated.

Returns

A value to use for the `instName` argument of IrisInstance::IrisInstance() or IrisInstance::registerInstance().

8.35.2.5 getParameter() [1/3]

```
const IrisValue & iris::IrisInstantiationContext::getParameter (
            const std::string & name )  [inline]
```

Get the value of an instantiation parameter as IrisValue.
This can be used as a fallback for all types not supported by the get<type>Parameter() functions below.

Parameters

name	The name of the parameter.

Returns

IrisValue of the parameter.

8.35.2.6 getParameter() [2/3]

```
void iris::IrisInstantiationContext::getParameter (
            const std::string & name,
            std::vector< uint64_t > & value )
```

Get the value of a large numeric instantiation parameter.
This is used for numeric parameters that are outside the range of uint64_t/int64_t.

Parameters

name	The name of the parameter.
value	A reference to a value of type *T* that receives the value of the named parameter.

8.35.2.7 getParameter() [3/3]

```
template<typename T >
void iris::IrisInstantiationContext::getParameter (
            const std::string & name,
            T & value )  [inline]
```

Get the value of an instantiation parameter.

Template Parameters

T	The type of the *value*. This must be a type that is appropriate to receive the value of this parameter.

Parameters

name	The name of the parameter.
value	A reference to a value of type *T* that receives the value of the named parameter.

8.35.2.8 getRecommendedInstanceFlags()

```
uint64_t iris::IrisInstantiationContext::getRecommendedInstanceFlags ( ) const  [inline]
```

Get the flags to use when registering the instance being instantiated.

Returns

A value to use for the flags argument of IrisInstance::IrisInstance() or IrisInstance::registerInstance().

8.35.2.9 getS64Parameter()

```
int64_t iris::IrisInstantiationContext::getS64Parameter (
            const std::string & name )  [inline]
```

Get the value of an instantiation parameter as int64_t.

Parameters

name	The name of the parameter.

Returns

S64 value.

8.35.2.10 getStringParameter()

```
std::string iris::IrisInstantiationContext::getStringParameter (
            const std::string & name )  [inline]
```

Get the value of an instantiation parameter as string.

Parameters

name	The name of the parameter.

Returns

String value.

8.35.2.11 getSubcomponentContext()

```
IrisInstantiationContext * iris::IrisInstantiationContext::getSubcomponentContext (
            const std::string & child_name )
```

Get an IrisInstanceContext pointer for a subcomponent instance.
For example, you might call getSubcomponentContext("cpu0") on the context "component.cluster0" to get the context to instantiate "component.cluster0.cpu0". The object pointed to by the return value is owned by its parent context and has the same lifetime as the parent context.

Parameters

child_name	The name of a child instance.

Returns

A pointer to an IrisInstantiationContext object for the named child.

8.35.2.12 getU64Parameter()

```
uint64_t iris::IrisInstantiationContext::getU64Parameter (
            const std::string & name )  [inline]
```

Get the value of an instantiation parameter as uint64_t.

Parameters

name	The name of the parameter.

Returns

U64 value.

8.35.2.13 parameterError()

```
void void void void iris::IrisInstantiationContext::parameterError (
            const std::string & code,
            const std::string & parameterName,
            const char * format,
             ... )
```

Add an error to the InstantiationResult.

See also

error

Parameters

code	An error code symbol. This should be one of the codes specified for the InstantiationError object.
parameterName	The name of the parameter this error relates to.
format	A printf-style format string.
...	Printf substitution arguments.

8.35.2.14 parameterWarning()

```
void void iris::IrisInstantiationContext::parameterWarning (
            const std::string & code,
            const std::string & parameterName,
            const char * format,
             ... )
```

Add a warning to the InstantiationResult.

See also

warning

Parameters

code	An error code symbol. This should be one of the codes specified for the InstantiationError object.
parameterName	The name of the parameter this warning relates to.
format	A printf-style format string.
...	Printf substitution arguments.

8.35.2.15 warning()

```
void iris::IrisInstantiationContext::warning (
            const std::string & code,
            const char * format,
             ... )
```

Add a warning to the InstantiationResult.

See also

parameterWarning

Parameters

code	An error code symbol. This should be one of the codes specified for the InstantiationError object.
format	A printf-style format string.
...	Printf substitution arguments.

The documentation for this class was generated from the following file:

- IrisInstantiationContext.h

8.36 iris::IrisNonFactoryPlugin< PLUGIN_CLASS > Class Template Reference

Wrapper to instantiate a non-factory plugin.
`#include <IrisPluginFactory.h>`

Public Member Functions

- **IrisNonFactoryPlugin** (IrisC_Functions *functions, const std::string &pluginName)

Static Public Member Functions

- static int64_t **initPlugin** (IrisC_Functions *functions, const std::string &pluginName)

8.36.1 Detailed Description

template<class PLUGIN_CLASS>
class iris::IrisNonFactoryPlugin< PLUGIN_CLASS >

Wrapper to instantiate a non-factory plugin.
Do not use this directly. Use the IRIS_NON_FACTORY_PLUGIN macro instead.

Template Parameters

PLUGIN_CLASS	Plugin class.

The documentation for this class was generated from the following file:

- IrisPluginFactory.h

8.37 iris::IrisParameterBuilder Class Reference

Helper class to construct instantiation parameters.
`#include <IrisParameterBuilder.h>`

Public Member Functions

- IrisParameterBuilder & addEnum (const std::string &symbol, const IrisValue &value, const std::string &description=std::string())

  Add an enum symbol for this parameter.

- IrisParameterBuilder & addStringEnum (const std::string &value, const std::string &description=std::string())

  Add a string enum symbol for this parameter.
- IrisParameterBuilder (ResourceInfo &info_)

  Construct a parameter builder for a given parameter resource.
- IrisParameterBuilder & setBitWidth (uint64_t bitWidth)

  Set the `bitWidth` field.
- IrisParameterBuilder & setDefault (const std::string &value)

  Set the default value for a string parameter.
- IrisParameterBuilder & setDefault (const std::vector< uint64_t > &value)

  Set the default value for a numeric parameter.
- IrisParameterBuilder & setDefault (uint64_t value)

  Set the default value for a numeric parameter.
- IrisParameterBuilder & setDefaultFloat (double value)

  Set the default value for a numericFp parameter.
- IrisParameterBuilder & setDefaultSigned (const std::vector< uint64_t > &value)

  Set the default value for a numericSigned parameter.
- IrisParameterBuilder & setDefaultSigned (int64_t value)

  Set the default value for a numericSigned parameter.
- IrisParameterBuilder & setDescr (const std::string &description)

  Set the `description` field.
- IrisParameterBuilder & setFormat (const std::string &format)

  Set the `format` field.
- IrisParameterBuilder & setHidden (bool hidden)

  Set the resource to hidden !
- IrisParameterBuilder & setInitOnly (bool value=true)

  Set the `initOnly` field.
- IrisParameterBuilder & setMax (const std::vector< uint64_t > &max)

  Set the `max` field.
- IrisParameterBuilder & setMax (uint64_t max)

  Set the `max` field.
- IrisParameterBuilder & setMaxFloat (double max)

  Set the `max` field for floating-point parameters.
- IrisParameterBuilder & setMaxSigned (const std::vector< uint64_t > &max)

  Set the `max` field.
- IrisParameterBuilder & setMaxSigned (int64_t max)

  Set the `max` field.
- IrisParameterBuilder & setMin (const std::vector< uint64_t > &min)

  Set the `min` field.
- IrisParameterBuilder & setMin (uint64_t min)

  Set the `min` field.
- IrisParameterBuilder & setMinFloat (double min)

  Set the `min` field for floating-point parameters.
- IrisParameterBuilder & setMinSigned (const std::vector< uint64_t > &min)

  Set the `min` field.
- IrisParameterBuilder & setMinSigned (int64_t min)

  Set the `min` field.
- IrisParameterBuilder & setName (const std::string &name)

  Set the `name` field.
- IrisParameterBuilder & setRange (const std::vector< uint64_t > &min, const std::vector< uint64_t > &max)

  Set both the `min` field and the `max` field.
- IrisParameterBuilder & setRange (uint64_t min, uint64_t max)

Set both the `min` field and the `max` field.

- IrisParameterBuilder & setRangeFloat (double min, double max)

  Set both the `min` field and the `max` field.
- IrisParameterBuilder & setRangeSigned (const std::vector< uint64_t > &min, const std::vector< uint64_t > &max)

  Set both the `min` field and the `max` field.
- IrisParameterBuilder & setRangeSigned (int64_t min, int64_t max)

  Set both the `min` field and the `max` field.
- IrisParameterBuilder & setRwMode (const std::string &rwMode)

  Set the `rwMode` field.
- IrisParameterBuilder & setSubRscId (uint64_t subRscId)

  Set the `subRscId` field.
- IrisParameterBuilder & setTag (const std::string &tag)

  Set a boolean tag for this parameter resource.
- IrisParameterBuilder & setTag (const std::string &tag, const IrisValue &value)

  Set a tag for this parameter resource.
- IrisParameterBuilder & setTopology (bool value=true)

  Set the `topology` field.
- IrisParameterBuilder & setType (const std::string &type)

  Set the type of this parameter.

8.37.1 Detailed Description

Helper class to construct instantiation parameters.

8.37.2 Constructor & Destructor Documentation

8.37.2.1 IrisParameterBuilder()

```
iris::IrisParameterBuilder::IrisParameterBuilder (
            ResourceInfo & info_ )  [inline]
```

Construct a parameter builder for a given parameter resource.

Parameters

*info_*	The resource info object for the parameter being built.

8.37.3 Member Function Documentation

8.37.3.1 addEnum()

```
IrisParameterBuilder & iris::IrisParameterBuilder::addEnum (
            const std::string & symbol,
            const IrisValue & value,
            const std::string & description = std::string() )  [inline]
```

Add an enum symbol for this parameter.

Parameters

symbol	The enum symbol that is being added.

Parameters

value	The value associated with the symbol.
description	A description explaining the meaning of the symbol.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.2 addStringEnum()

```
IrisParameterBuilder & iris::IrisParameterBuilder::addStringEnum (
            const std::string & value,
            const std::string & description = std::string() )  [inline]
```

Add a string enum symbol for this parameter.
For string enums, the symbol and value are the same.

Parameters

value	The value associated with the symbol.
description	A description explaining the meaning of the symbol.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.3 setBitWidth()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setBitWidth (
            uint64_t bitWidth )  [inline]
```

Set the `bitWidth` field.

Parameters

bitWidth	The `bitWidth` field of the ResourceInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.4 setDefault() [1/3]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setDefault (
            const std::string & value )  [inline]
```

Set the default value for a string parameter.

Parameters

value	The `defaultString` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.5 setDefault() [2/3]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setDefault (
            const std::vector< uint64_t > & value )  [inline]
```

Set the default value for a numeric parameter.
Use this variant for values that are >= 2**64.

Parameters

value	The `defaultData` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.6 setDefault() [3/3]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setDefault (
            uint64_t value )  [inline]
```

Set the default value for a numeric parameter.

Parameters

value	The `defaultData` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.7 setDefaultFloat()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setDefaultFloat (
            double value )  [inline]
```

Set the default value for a numericFp parameter.

Parameters

value	The `defaultData` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.8 setDefaultSigned() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setDefaultSigned (
            const std::vector< uint64_t > & value )  [inline]
```

Set the default value for a numericSigned parameter.

Use this variant for values that are out of range for int64_t.

Parameters

value	The `defaultData` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.9 setDefaultSigned() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setDefaultSigned (
            int64_t value ) [inline]
```

Set the default value for a numericSigned parameter.

Parameters

value	The `defaultData` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.10 setDescr()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setDescr (
            const std::string & description ) [inline]
```

Set the `description` field.

Parameters

description	The `description` field of the ResourceInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.11 setFormat()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setFormat (
            const std::string & format ) [inline]
```

Set the `format` field.

Parameters

format	The `format` field of the ResourceInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.12 setHidden()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setHidden (
            bool hidden ) [inline]
```

Set the resource to hidden !

Parameters

hidden	If true, this event source is not listed in resource_getList() calls but can still be accessed by resource_getResourceInfo() for clients that know the resource name. !

Returns

A reference to this TYPE object allowing calls to be chained together.

8.37.3.13 setInitOnly()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setInitOnly (
            bool value = true ) [inline]
```

Set the `initOnly` field.

Parameters

value	The `initOnly` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.14 setMax() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMax (
            const std::vector< uint64_t > & max ) [inline]
```

Set the `max` field.
Use this variant to set values that are >= 2**64.

Parameters

max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.15 setMax() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMax (
            uint64_t max ) [inline]
```

Set the `max` field.

Parameters

max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.16 setMaxFloat()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMaxFloat (
            double max )  [inline]
```

Set the `max` field for floating-point parameters.
This implies that the parameter type is "numericFp".

Parameters

max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.17 setMaxSigned() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMaxSigned (
            const std::vector< uint64_t > & max )  [inline]
```

Set the `max` field.
This implies that the parameter type is "numericSigned". Use this variant for signed values that are out of range for int64_t.

Parameters

max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.18 setMaxSigned() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMaxSigned (
            int64_t max )  [inline]
```

Set the `max` field.
This implies that the parameter type is "numericSigned".

Parameters

max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.19 setMin() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMin (
            const std::vector< uint64_t > & min )  [inline]
```

Set the `min` field.
Use this variant to set values that are >= 2**64.

Parameters

min	The `min` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.20 setMin() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMin (
            uint64_t min )  [inline]
```

Set the `min` field.

Parameters

min	The `min` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.21 setMinFloat()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMinFloat (
            double min )  [inline]
```

Set the `min` field for floating-point parameters.
This implies that the parameter type is "numericFp".

Parameters

min	The `min` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.22 setMinSigned() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMinSigned (
            const std::vector< uint64_t > & min )  [inline]
```

Set the `min` field.
This implies that the parameter type is "numericSigned". Use this variant for signed values that are out of range for int64_t.

Parameters

min	The `min` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.23 setMinSigned() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setMinSigned (
            int64_t min )  [inline]
```

Set the `min` field.
This implies that the parameter type is "numericSigned".

Parameters

min	The `min` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.24 setName()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setName (
            const std::string & name )  [inline]
```

Set the `name` field.

Parameters

name	The `name` field of the ResourceInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.25 setRange() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setRange (
            const std::vector< uint64_t > & min,
            const std::vector< uint64_t > & max )  [inline]
```

Set both the `min` field and the `max` field.
Use this variant to set values that are >= 2**64.

Parameters

min	The `min` field of the ParameterInfo object.
max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.26 setRange() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setRange (
            uint64_t min,
            uint64_t max )  [inline]
```

Set both the `min` field and the `max` field.

Parameters

min	The `min` field of the ParameterInfo object.
max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.27 setRangeFloat()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setRangeFloat (
            double min,
            double max )  [inline]
```

Set both the `min` field and the `max` field.
This implies that the parameter type is "numericFp".

Parameters

min	The `min` field of the ParameterInfo object.
max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.28 setRangeSigned() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setRangeSigned (
            const std::vector< uint64_t > & min,
            const std::vector< uint64_t > & max )  [inline]
```

Set both the `min` field and the `max` field.
This implies that the parameter type is "numericSigned". Use this variant for signed values that are out of range for int64_t.

Parameters

min	The `min` field of the ParameterInfo object.
max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.29 setRangeSigned() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setRangeSigned (
            int64_t min,
            int64_t max ) [inline]
```

Set both the `min` field and the `max` field.
This implies that the parameter type is "numericSigned".

Parameters

min	The `min` field of the ParameterInfo object.
max	The `max` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.30 setRwMode()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setRwMode (
            const std::string & rwMode ) [inline]
```

Set the `rwMode` field.

Parameters

rwMode	The `rwMode` field of the ResourceInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.31 setSubRscId()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setSubRscId (
            uint64_t subRscId ) [inline]
```

Set the `subRscId` field.

Parameters

subRscId	The `subRscId` field of the ResourceInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.32 setTag() [1/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setTag (
```

```
            const std::string & tag )  [inline]
```

Set a boolean tag for this parameter resource.

Parameters

tag	The name of the tag to set.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.33 setTag() [2/2]

```
IrisParameterBuilder & iris::IrisParameterBuilder::setTag (
            const std::string & tag,
            const IrisValue & value )  [inline]
```

Set a tag for this parameter resource.

Parameters

tag	The name of the tag to set.
value	The value to set for this tag.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.34 setTopology()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setTopology (
            bool value = true )  [inline]
```

Set the `topology` field.

Parameters

value	The `topology` field of the ParameterInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

8.37.3.35 setType()

```
IrisParameterBuilder & iris::IrisParameterBuilder::setType (
            const std::string & type )  [inline]
```

Set the type of this parameter.
The bitWidth field must be set before setting the type.

Parameters

type	The `type` field of the ResourceInfo object.

Returns

A reference to this IrisParameterBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisParameterBuilder.h

8.38 iris::IrisPluginFactory< PLUGIN_CLASS > Class Template Reference

Public Member Functions

- **IrisPluginFactory** (IrisC_Functions *iris_c_functions, const std::string &plugin_name)
- IrisErrorCode **unregisterInstance** ()

Static Public Member Functions

- static int64_t **initPlugin** (IrisC_Functions *functions, const std::string &plugin_name)

The documentation for this class was generated from the following file:

- IrisPluginFactory.h

8.39 iris::IrisPluginFactoryBuilder Class Reference

Set meta data for instantiating a plug-in instance.
`#include <IrisPluginFactory.h>`
Inherits iris::IrisInstanceFactoryBuilder.

Public Member Functions

- const std::string & getDefaultInstanceName () const
  Get the default name to use for plug-in instances.
- const std::string & getInstanceNamePrefix () const
  Get the prefix to use for instances of this plug-in.
- const std::string & getPluginName () const
  Get the plug-in name.
- IrisPluginFactoryBuilder (const std::string &name)
- void setDefaultInstanceName (const std::string &name)
  Override the default instance name for plug-in instances.
- void setInstanceNamePrefix (const std::string &prefix)
  Override the instance name prefix. The default is "client.plugin".
- void setPluginName (const std::string &name)
  Override the plug-in name.

8.39.1 Detailed Description

Set meta data for instantiating a plug-in instance.

8.39.2 Constructor & Destructor Documentation

8.39.2.1 IrisPluginFactoryBuilder()

```
iris::IrisPluginFactoryBuilder::IrisPluginFactoryBuilder (
            const std::string & name )  [inline]
```

Parameters

name	The name of the plug-in to build.

8.39.3 Member Function Documentation

8.39.3.1 getDefaultInstanceName()

```
const std::string & iris::IrisPluginFactoryBuilder::getDefaultInstanceName ( ) const  [inline]
```

Get the default name to use for plug-in instances.

Returns

The default name for plug-in instances.

8.39.3.2 getInstanceNamePrefix()

```
const std::string & iris::IrisPluginFactoryBuilder::getInstanceNamePrefix ( ) const  [inline]
```

Get the prefix to use for instances of this plug-in.

Returns

The prefix to use for instances of this plug-in.

8.39.3.3 getPluginName()

```
const std::string & iris::IrisPluginFactoryBuilder::getPluginName ( ) const  [inline]
```

Get the plug-in name.

Returns

The name of the plug-in.

8.39.3.4 setDefaultInstanceName()

```
void iris::IrisPluginFactoryBuilder::setDefaultInstanceName (
            const std::string & name )  [inline]
```

Override the default instance name for plug-in instances.
The factory provides a sensible default for this name so it should only be overridden if there is a good reason to do so.

Parameters

name	The default name for plug-in instances.

8.39.3.5 setInstanceNamePrefix()

```
void iris::IrisPluginFactoryBuilder::setInstanceNamePrefix (
            const std::string & prefix )  [inline]
```

Override the instance name prefix. The default is "client.plugin".

The factory provides a sensible default for this prefix so it should only be overridden if there is a good reason to do so.

Parameters

prefix	The prefix that will be used for instances of this plug-in.

8.39.3.6 setPluginName()

```
void iris::IrisPluginFactoryBuilder::setPluginName (
            const std::string & name )  [inline]
```

Override the plug-in name.
The factory provides a sensible default for this name so it should only be overridden if there is a good reason to do so.

Parameters

name	The name of the plug-in.

The documentation for this class was generated from the following file:

- IrisPluginFactory.h

8.40 iris::IrisRegisterReadEventEmitter< REG_T, ARGS > Class Template Reference

An EventEmitter class for register read events.
`#include <IrisRegisterEventEmitter.h>`
Inherits IrisRegisterEventEmitterBase.

Public Member Functions

- void operator() (ResourceId rscId, bool debug, REG_T value, ARGS... args)

  Emit an event.

8.40.1 Detailed Description

template<typename REG_T, typename... ARGS>
class iris::IrisRegisterReadEventEmitter< REG_T, ARGS >

An EventEmitter class for register read events.

Template Parameters

REG_T	The type of the register being read.
ARGS	The types of any custom fields that this event source defines, in addition to the standard fields defined for register read events.

Use IrisRegisterReadEventEmitter with IrisInstanceBuilder to add register read events to your Iris instance:

```
// Declare an event emitter
iris::IrisRegisterReadEventEmitter<uint64_t> reg_read_event;
// Add it to an Iris instance
iris::IrisInstance my_instance(...);
iris::IrisInstanceBuilder *builder = my_instance->getBuilder();
builder->setRegisterReadEvent("READ_REG", reg_read_event);
// Add some registers that will be traced by this event
```

```
builder->setNextRscId(0x1000);
builder->addRegister("X0", 64, "Register X0");
builder->addRegister("X1", 64, "Register X1");
builder->addRegister("X2", 64, "Register X2");
builder->addRegister("X3", 64, "Register X3");
// Now that the Instance builder has the metadata for the registers, we need
// to finalize the register read event to populate the event metadata.
builder->finalizeRegisterReadEvent();
uint64_t readRegister(unsigned reg_index, bool is_debug)
{
     uint64_t value = readRegValue(reg_index);
     // Emit an event
     reg_read_event(0x1000 | reg_index, is_debug, value);
     return value;
 }
```

8.40.2 Member Function Documentation

8.40.2.1 operator()()

```
template<typename REG_T , typename...  ARGS>
void iris::IrisRegisterReadEventEmitter< REG_T, ARGS >::operator() (
             ResourceId rscId,
             bool debug,
             REG_T value,
             ARGS...  args )  [inline]
```

Emit an event.

Parameters

rscId	Resource id for the register that was accessed.
debug	True if this access originated from a debug access.
value	The register value that was read during this event.
args	Any additional custom fields for this event.

The documentation for this class was generated from the following file:

- IrisRegisterEventEmitter.h

8.41 iris::IrisRegisterUpdateEventEmitter< REG_T, ARGS > Class Template Reference

An EventEmitter class for register update events.
`#include <IrisRegisterEventEmitter.h>`
Inherits IrisRegisterEventEmitterBase.

Public Member Functions

- void operator() (ResourceId rscId, bool debug, REG_T old_value, REG_T new_value, ARGS... args)

  Emit an event.

8.41.1 Detailed Description

template<typename REG_T, typename... ARGS>
class iris::IrisRegisterUpdateEventEmitter< REG_T, ARGS >

An EventEmitter class for register update events.

Template Parameters

REG_T	The type of the register being read.
ARGS	Types of any custom fields that this event source defines, in addition to the standard fields defined for register update events.

Use IrisRegisterUpdateEventEmitter with IrisInstanceBuilder to add register update events to your Iris instance:

```
// Declare an event emitter
iris::IrisRegisterUpdateEventEmitter<uint64_t> reg_update_event;
// Add it to an Iris instance
iris::IrisInstance my_instance(...);
iris::IrisInstanceBuilder *builder = my_instance->getBuilder();
builder->setRegisterUpdateEvent("WRITE_REG", reg_update_event);
// Add some registers that will be traced by this event
builder->setNextRscId(0x1000);
builder->addRegister("X0", 64, "Register X0");
builder->addRegister("X1", 64, "Register X1");
builder->addRegister("X2", 64, "Register X2");
builder->addRegister("X3", 64, "Register X3");
// Now that the Instance builder has the metadata for the registers, we need
// to finalize the register update event to populate the event metadata.
builder->finalizeRegisterUpdateEvent();
void writeRegister(unsigned reg_index, bool is_debug, uint64_t new_value)
{
     uint64_t old_value = readRegValue(reg_index);
     writeRegValue(reg_index, new_value);
     // Emit an event
     reg_update_event(0x1000 | reg_index, is_debug, old_value, new_value);
 }
```

8.41.2 Member Function Documentation

8.41.2.1 operator()()

```
template<typename REG_T , typename...  ARGS>
void iris::IrisRegisterUpdateEventEmitter< REG_T, ARGS >::operator() (
             ResourceId rscId,
             bool debug,
             REG_T old_value,
             REG_T new_value,
             ARGS...  args )  [inline]
```

Emit an event.

Parameters

rscId	Resource id for the register that was accessed.
debug	True if this access originated from a debug access.
old_value	The register value before the event.
new_value	The register value after the event.
args	Any additional custom fields for this event.

The documentation for this class was generated from the following file:

- IrisRegisterEventEmitter.h

8.42 iris::IrisSimulationResetContext Class Reference

Provides context to a reset delegate call.

`#include <IrisInstanceSimulation.h>`

Public Member Functions

- bool getAllowPartialReset () const

  Get the allowPartialReset flag.
- void **setAllowPartialReset** (bool value=true)

8.42.1 Detailed Description

Provides context to a reset delegate call.

8.42.2 Member Function Documentation

8.42.2.1 getAllowPartialReset()

```
bool iris::IrisSimulationResetContext::getAllowPartialReset ( ) const  [inline]
```

Get the allowPartialReset flag.

Returns

Returns true if simulation_reset() was called with allowPartialReset=true.

The documentation for this class was generated from the following file:

- IrisInstanceSimulation.h

8.43 iris::IrisInstanceBuilder::MemorySpaceBuilder Class Reference

Used to set metadata for a memory space.

`#include <IrisInstanceBuilder.h>`

Public Member Functions

- MemorySpaceBuilder & addAttribute (const std::string &name, AttributeInfo attrib)

  Add an attribute to the `attrib` field.
- MemorySpaceId getSpaceId () const

  Get the memory space id for this memory space.
- **MemorySpaceBuilder** (IrisInstanceMemory::SpaceInfoAndAccess &info_)
- MemorySpaceBuilder & setAttributeDefault (const std::string &name, IrisValue value)

  Set the default value for an attribute in the `attrib` field.
- MemorySpaceBuilder & setAttributes (const AttributeInfoMap &attribInfoMap)

  Add attributes to the `attrib` field.
- MemorySpaceBuilder & setCanonicalMsn (uint64_t canonicalMsn)

  Set the `canonicalMsn` field.
- MemorySpaceBuilder & setDescription (const std::string &description)

  Set the `description` field.
- MemorySpaceBuilder & setEndianness (const std::string &endianness)

  Set the `endianness` field.
- MemorySpaceBuilder & setMaxAddr (uint64_t maxAddr)

  Set the `maxAddr` field.
- MemorySpaceBuilder & setMinAddr (uint64_t minAddr)

  Set the `minAddr` field.
- MemorySpaceBuilder & setName (const std::string &name)

  Set the `name` field.

- template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) FUNC>
  MemorySpaceBuilder & setReadDelegate ()

  Set the delegate to read this memory space.
- MemorySpaceBuilder & setReadDelegate (MemoryReadDelegate delegate)

  Set the delegate to read this memory space.
- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult &) METHOD>
  MemorySpaceBuilder & setReadDelegate (T *instance)

  Set the delegate to read this memory space.
- template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std::vector< std::string > &, IrisValueMap &) FUNC>
  MemorySpaceBuilder & setSidebandDelegate ()

  Set the delegate to read sideband information.
- MemorySpaceBuilder & setSidebandDelegate (MemoryGetSidebandInfoDelegate delegate)

  Set the delegate to read sideband information.
- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std::vector< std::string > &, IrisValueMap &) METHOD>
  MemorySpaceBuilder & setSidebandDelegate (T *instance)

  Set the delegate to read sideband information.
- MemorySpaceBuilder & setSupportedByteWidths (uint64_t supportedByteWidths)

  Set the `supportedByteWidths` field.
- template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) FUNC>
  MemorySpaceBuilder & setWriteDelegate ()

  Set the delegate to write to this memory space.
- MemorySpaceBuilder & setWriteDelegate (MemoryWriteDelegate delegate)

  Set the delegate to write to this memory space.
- template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) METHOD>
  MemorySpaceBuilder & setWriteDelegate (T *instance)

  Set the delegate to write to this memory space.

8.43.1 Detailed Description

Used to set metadata for a memory space.

8.43.2 Member Function Documentation

8.43.2.1 addAttribute()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::addAttribute (
            const std::string & name,
            AttributeInfo attrib ) [inline]
```

Add an attribute to the `attrib` field.

Parameters

name	The name of this attribute.
attrib	AttributeInfo for this attribute.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.2 getSpaceId()

```
MemorySpaceId iris::IrisInstanceBuilder::MemorySpaceBuilder::getSpaceId ( ) const  [inline]
```

Get the memory space id for this memory space.
This can be useful for setting up address translations and to map access requests to the correct memory space in memory access delegates.

Returns

The memory space id for this memory space.

8.43.2.3 setAttributeDefault()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setAttributeDefault (
            const std::string & name,
            IrisValue value )  [inline]
```

Set the default value for an attribute in the `attrib` field.

Parameters

name	The name of this attribute.
value	Default value of the named attribute.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.4 setAttributes()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setAttributes (
            const AttributeInfoMap & attribInfoMap )  [inline]
```

Add attributes to the `attrib` field.

Parameters

attribInfoMap	The attributes of this memory space.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.5 setCanonicalMsn()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setCanonicalMsn (
            uint64_t canonicalMsn )  [inline]
```

Set the `canonicalMsn` field.

Parameters

canonicalMsn	The canonicalMsn field of the MemorySpaceInfo object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.6 setDescription()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setDescription (
            const std::string & description )  [inline]
```

Set the `description` field.

Parameters

description	The description field of the MemorySpaceInfo object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.7 setEndianness()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setEndianness (
            const std::string & endianness )  [inline]
```

Set the `endianness` field.

Parameters

endianness	The endianness field of the MemorySpaceInfo object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.8 setMaxAddr()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setMaxAddr (
            uint64_t maxAddr )  [inline]
```

Set the `maxAddr` field.

Parameters

maxAddr	The maxAddr field of the MemorySpaceInfo object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.9 setMinAddr()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setMinAddr (
            uint64_t minAddr )  [inline]
```

Set the `minAddr` field.

Parameters

minAddr	The minAddr field of the MemorySpaceInfo object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.10 setName()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setName (
            const std::string & name )  [inline]
```

Set the `name` field.

Parameters

name	The name field of the MemorySpaceInfo object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.11 setReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const Attribute↩
ValueMap &, MemoryReadResult &) FUNC>
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setReadDelegate ( )  [inline]
```

Set the delegate to read this memory space.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultMemoryReadDelegate

Template Parameters

FUNC	A memory read delegate function.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.12 setReadDelegate() [2/3]

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setReadDelegate (
            MemoryReadDelegate delegate )  [inline]
```

Set the delegate to read this memory space.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultMemoryReadDelegate

Parameters

delegate	MemoryReadDelegate object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.13 setReadDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64↩
_t, const AttributeValueMap &, MemoryReadResult &) METHOD>
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setReadDelegate (
            T * instance )  [inline]
```

Set the delegate to read this memory space.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultMemoryReadDelegate

Template Parameters

T	A class that defines a method with the right signature to be a memory read delegate.
METHOD	A memory read delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.14 setSidebandDelegate() [1/3]

```
template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std↩
::vector< std::string > &, IrisValueMap &) FUNC>
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setSidebandDelegate ( )
[inline]
```

Set the delegate to read sideband information.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate

Template Parameters

FUNC	A memory sideband information delegate function.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.15 setSidebandDelegate() [2/3]

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setSidebandDelegate (
            MemoryGetSidebandInfoDelegate delegate ) [inline]
```

Set the delegate to read sideband information.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate

Parameters

delegate	MemoryGetSidebandInfoDelegate object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.16 setSidebandDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, const IrisValue↩
Map &, const std::vector< std::string > &, IrisValueMap &) METHOD>
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setSidebandDelegate (
            T * instance ) [inline]
```

Set the delegate to read sideband information.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultGetMemorySidebandInfoDelegate

Template Parameters

T	A class that defines a method with the right signature to be a memory sideband information delegate.
METHOD	A memory sideband information delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.17 setSupportedByteWidths()

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setSupportedByteWidths (
            uint64_t supportedByteWidths ) [inline]
```

Set the `supportedByteWidths` field.
Usage:
setSupportedByteWidths(1+2+4+8+16); // Indicate support for byteWidth 1, 2, 4, 8, and 16.

Parameters

supportedByteWidths	Outer envelope of all supported byteWidth values Bit mask: Bit N==1 means byteWidth 1 << N is supported.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.18 setWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) FUNC>
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setWriteDelegate ( )  [inline]
```

Set the delegate to write to this memory space.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultMemoryWriteDelegate

Template Parameters

FUNC	A memory write delegate function.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.19 setWriteDelegate() [2/3]

```
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setWriteDelegate (
            MemoryWriteDelegate delegate )  [inline]
```

Set the delegate to write to this memory space.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultMemoryWriteDelegate

Parameters

delegate	MemoryWriteDelegate object.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

8.43.2.20 setWriteDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const MemorySpaceInfo &, uint64_t, uint64_t, uint64↩
_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult &) METHOD>
MemorySpaceBuilder & iris::IrisInstanceBuilder::MemorySpaceBuilder::setWriteDelegate (
            T * instance )  [inline]
```

Set the delegate to write to this memory space.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultMemoryWriteDelegate

Template Parameters

T	A class that defines a method with the right signature to be a memory write delegate.
METHOD	A memory write delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this MemorySpaceBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.44 iris::IrisCommandLineParser::Option Struct Reference

Option container.
`#include <IrisCommandLineParser.h>`

Public Member Functions

- Option & setList (char sep=',')

Friends

- class **IrisCommandLineParser**

8.44.1 Detailed Description

Option container.

8.44.2 Member Function Documentation

8.44.2.1 setList()

```
Option & iris::IrisCommandLineParser::Option::setList (
            char sep = ',' )  [inline]
```

Make this option a "list" option which can be specified multiple times. The value is stored as a single string and the elements are separated by "sep". Use getList() or getMap() to extract the elements.
The documentation for this struct was generated from the following file:

- IrisCommandLineParser.h

8.45 iris::IrisInstanceBuilder::ParameterBuilder Class Reference

Used to set metadata on a parameter.
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- ParameterBuilder & addEnum (const std::string &symbol, const IrisValue &value, const std::string &description=std::string())

  Add a symbol to the enums field for numeric resources.
- ParameterBuilder & addStringEnum (const std::string &stringValue, const std::string &description=std::string())

  Add a symbol to the enums field for string resources.
- ResourceId getRscId () const

  Return the rscId that was allocated for this resource.
- ParameterBuilder & getRscId (ResourceId &rscIdOut)

  Get the rscId that was allocated for this resource.
- **ParameterBuilder** (IrisInstanceResource::ResourceInfoAndAccess &info_)
- ParameterBuilder & setBitWidth (uint64_t bitWidth)

  Set the `bitWidth` field.
- ParameterBuilder & setCname (const std::string &cname)

  Set the `cname` field.
- template<typename T >
  ParameterBuilder & setDefaultData (std::initializer_list< T > &&t)

  Set the `default` value for wide numeric parameters.
- ParameterBuilder & setDefaultData (uint64_t value)

  Set the `default` value for numeric parameter to a value <= 64 bit.
- template<typename Container >
  ParameterBuilder & setDefaultDataFromContainer (const Container &container)

  Set the `default` value for wide numeric parameters.
- ParameterBuilder & setDefaultString (const std::string &defaultString)

  Set the `defaultData` field for wide numeric parameters (bitWidth > 64 bit).
- ParameterBuilder & **setDescr** (const std::string &description)

  Deprecated alias for setDescription().
- ParameterBuilder & setDescription (const std::string &description)

  Set the `description` field.
- ParameterBuilder & setFormat (const std::string &format)

  Set the `format` field.
- ParameterBuilder & setHidden (bool hidden=true)

  Set the resource to hidden.
- ParameterBuilder & setInitOnly (bool initOnly=true)

  Set the `initOnly` flag of a parameter.
- template<typename T >
  ParameterBuilder & setMax (std::initializer_list< T > &&t)

Set the `max` field for wide numeric parameters.

- ParameterBuilder & setMax (uint64_t value)

  Set the `max` field to a value <= 64 bit.

- template<typename Container >
  ParameterBuilder & setMaxFromContainer (const Container &container)

  Set the `max` field for wide numeric parameters.

- template<typename T >
  ParameterBuilder & setMin (std::initializer_list< T > &&t)

  Set the `min` field for wide numeric parameters.

- ParameterBuilder & setMin (uint64_t value)

  Set the `min` field to a value <= 64 bit.

- template<typename Container >
  ParameterBuilder & setMinFromContainer (const Container &container)

  Set the `min` field for wide numeric parameters.

- ParameterBuilder & setName (const std::string &name)

  Set the `name` field.

- ParameterBuilder & setParentRscId (ResourceId parentRscId)

  Set the `parentRscId` field.

- template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
  ParameterBuilder & setReadDelegate ()

  Set the delegate to read the resource.

- ParameterBuilder & setReadDelegate (ResourceReadDelegate readDelegate)

  Set the delegate to read the resource.

- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
  ParameterBuilder & setReadDelegate (T *instance)

  Set the delegate to read the resource.

- ParameterBuilder & setRwMode (const std::string &rwMode)

  Set the `rwMode` field.

- ParameterBuilder & setSubRscId (uint64_t subRscId)

  Set the `subRscId` field.

- ParameterBuilder & setTag (const std::string &tag)

  Set the named boolean tag to true (e.g. isPc)

- ParameterBuilder & setTag (const std::string &tag, const IrisValue &value)

  Set a tag to the specified value.

- ParameterBuilder & setType (const std::string &type)

  Set the `type` field.

- template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
  ParameterBuilder & setWriteDelegate ()

  Set the delegate to write the resource.

- ParameterBuilder & setWriteDelegate (ResourceWriteDelegate writeDelegate)

  Set the delegate to write the resource.

- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &) METHOD>
  ParameterBuilder & setWriteDelegate (T *instance)

  Set the delegate to write the resource.

8.45.1 Detailed Description

Used to set metadata on a parameter.

8.45.2 Member Function Documentation

8.45.2.1 addEnum()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::addEnum (
            const std::string & symbol,
            const IrisValue & value,
            const std::string & description = std::string() )  [inline]
```

Add a symbol to the enums field for numeric resources.
This should be called multiple times to add multiple symbols.

Parameters

symbol	The symbol string to be associated with the specified value.
value	The value of this symbol.
description	A description of this symbol.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.2 addStringEnum()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::addStringEnum (
            const std::string & stringValue,
            const std::string & description = std::string() )  [inline]
```

Add a symbol to the enums field for string resources.
This should be called multiple times to add multiple symbols.

Parameters

value	The string value of this symbol. This is also used as the symbols string.
description	A description of this symbol.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.3 getRscId() [1/2]

```
ResourceId iris::IrisInstanceBuilder::ParameterBuilder::getRscId ( ) const  [inline]
```

Return the rscId that was allocated for this resource.

Returns

The rscId that was allocated for this resource.

8.45.2.4 getRscId() [2/2]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::getRscId (
            ResourceId & rscIdOut )  [inline]
```

Get the rscId that was allocated for this resource.
This variant is useful to get the ResourceId of fields added in a chained call
where return values are not practical.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.5 setBitWidth()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setBitWidth (
            uint64_t bitWidth )  [inline]
```

Set the `bitWidth` field.

Parameters

bitWidth	The bitWidth field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.6 setCname()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setCname (
            const std::string & cname )  [inline]
```

Set the `cname` field.

Parameters

cname	The cname field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.7 setDefaultData() [1/2]

```
template<typename T >
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setDefaultData (
            std::initializer_list< T > && t )  [inline]
```

Set the `default` value for wide numeric parameters.
This function accepts a braced initializer-list and is otherwise idential to setDefaultDataFromContainer().
Each element will be promoted/narrowed to uint64_t.

Parameters

t	Braced initializer-list.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.8 setDefaultData() [2/2]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setDefaultData (
```

```
            uint64_t value )  [inline]
```

Set the `default` value for numeric parameter to a value <= 64 bit.
If the parameter is wider than the passed value the value is zero extended.
If the parameter is narrower than the passed value the superfluous bits are ignored.

Parameters

value	The defaultData field of the ParameterInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.9 setDefaultDataFromContainer()

```
template<typename Container >
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setDefaultDataFromContainer (
            const Container & container )  [inline]
```

Set the `default` value for wide numeric parameters.
Container must be a type which allows to iterate over uint64_t bit chunks of the value,
least significant bits first, for example std::array<uint64_t> or std::vector<uint64_t>.
Each element of the container will be promoted/narrowed to uint64_t.
If the parameter is wider than the passed value the value is zero extended.
If the parameter is narrower than the passed value the superfluous bits are ignored.

Parameters

container	Container containing the value in 64-bit chunks.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.10 setDefaultString()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setDefaultString (
            const std::string & defaultString )  [inline]
```

Set the `defaultData` field for wide numeric parameters (bitWidth > 64 bit).
Set the default value for string parameters.

Parameters

defaultString	The defaultString field of the ParameterInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.11 setDescription()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setDescription (
            const std::string & description )  [inline]
```

Set the `description` field.

Parameters

description	The description field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.12 setFormat()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setFormat (
            const std::string & format ) [inline]
```

Set the `format` field.

Parameters

format	The format field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.13 setHidden()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setHidden (
            bool hidden = true ) [inline]
```

Set the resource to hidden.

Parameters

hidden	If true, this resource is not listed in resource_getList() calls

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.14 setInitOnly()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setInitOnly (
            bool initOnly = true ) [inline]
```

Set the `initOnly` flag of a parameter.
This also implicitly sets the parameter to read-only.

Parameters

initOnly	The initOnly flag of a parameter.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.15 setMax() [1/2]

```
template<typename T >
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setMax (
            std::initializer_list< T > && t )  [inline]
```

Set the max field for wide numeric parameters.
This function accepts a braced initializer-list and is otherwise idential to setMaxFromContainer().
Each element will be promoted/narrowed to uint64_t.

Parameters

t	Braced initializer-list.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.16 setMax() [2/2]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setMax (
            uint64_t value )  [inline]
```

Set the max field to a value <= 64 bit.
If the parameter is wider than the passed value the value is zero extended.
If the parameter is narrower than the passed value the superfluous bits are ignored.

Parameters

value	Max value of the parameter.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.17 setMaxFromContainer()

```
template<typename Container >
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setMaxFromContainer (
            const Container & container )  [inline]
```

Set the max field for wide numeric parameters.
Container must be a type which allows to iterate over uint64_t bit chunks of the value,
least significant bits first, for example std::array<uint64_t> or std::vector<uint64_t>.
Each element of the container will be promoted/narrowed to uint64_t.
If the parameter is wider than the passed value the value is zero extended.
If the parameter is narrower than the passed value the superfluous bits are ignored.

Parameters

container	Container containing the value in 64-bit chunks.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.18 setMin() [1/2]

```
template<typename T >
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setMin (
            std::initializer_list< T > && t )  [inline]
```

Set the min field for wide numeric parameters.
This function accepts a braced initializer-list and is otherwise idential to setMinFromContainer().
Each element will be promoted/narrowed to uint64_t.

Parameters

t	Braced initializer-list.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.19 setMin() [2/2]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setMin (
            uint64_t value )  [inline]
```

Set the min field to a value <= 64 bit.
If the parameter is wider than the passed value the value is zero extended.
If the parameter is narrower than the passed value the superfluous bits are ignored.

Parameters

value	min value of the parameter.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.20 setMinFromContainer()

```
template<typename Container >
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setMinFromContainer (
            const Container & container )  [inline]
```

Set the min field for wide numeric parameters.
Container must be a type which allows to iterate over uint64_t bit chunks of the value,
least significant bits first, for example std::array<uint64_t> or std::vector<uint64_t>.
Each element of the container will be promoted/narrowed to uint64_t.
If the parameter is wider than the passed value the value is zero extended.
If the parameter is narrower than the passed value the superfluous bits are ignored.

Parameters

container	Container containing the value in 64-bit chunks.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.21 setName()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setName (
            const std::string & name )  [inline]
```

Set the `name` field.

Parameters

name	The name field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.22 setParentRscId()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setParentRscId (
            ResourceId parentRscId )  [inline]
```

Set the `parentRscId` field.
This function makes this register a child of the specified parent. It is not necessary to call this function when adding child registers using the addField() function.

Parameters

parent↩ RscId	The rscId of the parent register.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.23 setReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setReadDelegate ( )  [inline]
```

Set the delegate to read the resource.
Set a delegate which calls function FUNC().
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Template Parameters

FUNC	A resource read delegate function.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.24 setReadDelegate() [2/3]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setReadDelegate (
```

```
ResourceReadDelegate readDelegate )  [inline]
```

Set the delegate to read the resource.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Parameters

readDelegate	ResourceReadDelegate object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.25 setReadDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setReadDelegate (
            T * instance )  [inline]
```

Set the delegate to read the resource.
Set a delegate which calls METHOD() on an instance of class T.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Template Parameters

T	A class that defines a method with the right signature to be a resource read delegate.
METHOD	A resource read delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.26 setRwMode()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setRwMode (
            const std::string & rwMode )  [inline]
```

Set the `rwMode` field.

Parameters

rwMode	The rwMode field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.27 setSubRscId()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setSubRscId (
            uint64_t subRscId )  [inline]
```

Set the `subRscId` field.

Parameters

subRscId	The subRscId field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.28 setTag() [1/2]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setTag (
            const std::string & tag )  [inline]
```

Set the named boolean tag to true (e.g. isPc)

Parameters

tag	The name of the tag to set.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.29 setTag() [2/2]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setTag (
            const std::string & tag,
            const IrisValue & value )  [inline]
```

Set a tag to the specified value.

Parameters

tag	The name of the tag to set.
value	The value to set the tag to.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.30 setType()

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setType (
            const std::string & type )  [inline]
```

Set the `type` field.

Parameters

type	The type field of the ResourceInfo object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.31 setWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setWriteDelegate ( )  [inline]
```

Set the delegate to write the resource.
Set a delegate which calls function FUNC().
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Template Parameters

FUNC	A resource write delegate function.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.32 setWriteDelegate() [2/3]

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setWriteDelegate (
            ResourceWriteDelegate writeDelegate )  [inline]
```

Set the delegate to write the resource.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Parameters

writeDelegate	ResourceWriteDelegate object.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

8.45.2.33 setWriteDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &)
METHOD>
```

```
ParameterBuilder & iris::IrisInstanceBuilder::ParameterBuilder::setWriteDelegate (
            T * instance )  [inline]
```

Set the delegate to write the resource.
Set a delegate which calls METHOD() on an instance of class T.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Template Parameters

T	A class that defines a method with the right signature to be a resource write delegate.
METHOD	A resource write delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this ParameterBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.46 iris::IrisInstanceEvent::ProxyEventInfo Struct Reference

Contains information for a single proxy EventSource.
`#include <IrisInstanceEvent.h>`

Public Attributes

- std::vector< EventStreamId > **evStreamIds**
- EventSourceId **targetEvSrcId** {}
- InstanceId **targetInstId** {}

8.46.1 Detailed Description

Contains information for a single proxy EventSource.
The documentation for this struct was generated from the following file:

- IrisInstanceEvent.h

8.47 iris::IrisInstanceBuilder::RegisterBuilder Class Reference

Used to set metadata on a register resource.
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- RegisterBuilder & addEnum (const std::string &symbol, const IrisValue &value, const std::string &description=std::string())

  Add a symbol to the enums field for numeric resources.

- FieldBuilder addField (const std::string &name, uint64_t lsbOffset, uint64_t bitWidth, const std::string &description)

  Add a subregister field to this register. By default, the field copies attributes from its parent register, but any field can be overridden.
- FieldBuilder addLogicalField (const std::string &name, uint64_t bitWidth, const std::string &description)

  Add a logical subregister field to this register. A logical field is a field which has a bitwidth, but which does not have an lsbOffset. It is usually used to represent non-contiguous fields which are distributed across multiple chunks in the parent register as a single contiguous register. This allows to attach enums to such a field.
- RegisterBuilder & addStringEnum (const std::string &stringValue, const std::string &description=std::string())

  Add a symbol to the enums field for string resources.
- ResourceId getRscId () const

  Return the rscId that was allocated for this resource.
- RegisterBuilder & getRscId (ResourceId &rscIdOut)

  Get the rscId that was allocated for this resource.
- **RegisterBuilder** (IrisInstanceResource::ResourceInfoAndAccess &info_, IrisInstanceResource *inst_↩ resource_, IrisInstanceBuilder *instance_builder_)
- RegisterBuilder & setAddressOffset (uint64_t addressOffset)

  Set the `addressOffset` field.
- RegisterBuilder & setBitWidth (uint64_t bitWidth)

  Set the `bitWidth` field.
- RegisterBuilder & setBreakpointSupportInfo (const std::string &supported)

  Set the `breakpointSupport` field.
- RegisterBuilder & setCanonicalRn (uint64_t canonicalRn_)

  Set the `canonicalRn` field.
- RegisterBuilder & setCanonicalRnElfDwarf (uint16_t architecture, uint16_t dwarfRegNum)

  Set the `canonicalRn` field for "ElfDwarf" scheme.
- RegisterBuilder & setCname (const std::string &cname)

  Set the `cname` field.
- RegisterBuilder & **setDescr** (const std::string &description)

  Deprecated alias for setDescription().
- RegisterBuilder & setDescription (const std::string &description)

  Set the `description` field.
- RegisterBuilder & setFormat (const std::string &format)

  Set the `format` field.
- RegisterBuilder & setLsbOffset (uint64_t lsbOffset)

  Set the `lsbOffset` field.
- RegisterBuilder & setName (const std::string &name)

  Set the `name` field.
- RegisterBuilder & setParentRscId (ResourceId parentRscId)

  Set the `parentRscId` field.
- template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
  RegisterBuilder & setReadDelegate ()

  Set the delegate to read the resource.
- RegisterBuilder & setReadDelegate (ResourceReadDelegate readDelegate)

  Set the delegate to read the resource.
- template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
  RegisterBuilder & setReadDelegate (T *instance)

  Set the delegate to read the resource.
- template<typename T >
  RegisterBuilder & setResetData (std::initializer_list< T > &&t)

  Set the `resetData` field for wide registers.
- RegisterBuilder & setResetData (uint64_t value)

Set the `resetData` field to a value <= 64 bit.

- template<typename Container >
  RegisterBuilder & setResetDataFromContainer (const Container &container)

  Set the `resetData` field for wide registers.
- RegisterBuilder & setResetString (const std::string &resetString)

  Set the `resetString` field.
- RegisterBuilder & setRwMode (const std::string &rwMode)

  Set the `rwMode` field.
- RegisterBuilder & setSubRscId (uint64_t subRscId)

  Set the `subRscId` field.
- RegisterBuilder & setTag (const std::string &tag)

  Set the named boolean tag to true (e.g. isPc)
- RegisterBuilder & setTag (const std::string &tag, const IrisValue &value)

  Set a tag to the specified value.
- RegisterBuilder & setType (const std::string &type)

  Set the `type` field.
- template<IrisErrorCode(∗)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
  RegisterBuilder & setWriteDelegate ()

  Set the delegate to write the resource.
- RegisterBuilder & setWriteDelegate (ResourceWriteDelegate writeDelegate)

  Set the delegate to write the resource.
- template<typename T , IrisErrorCode(T::∗)(const ResourceInfo &, const ResourceWriteValue &) METHOD>
  RegisterBuilder & setWriteDelegate (T ∗instance)

  Set the delegate to write the resource.
- template<typename T >
  RegisterBuilder & setWriteMask (std::initializer_list< T > &&t)

  Set the `writeMask` field for wide registers.
- RegisterBuilder & setWriteMask (uint64_t value)

  Set the `writeMask` field to a value <= 64 bit.
- template<typename Container >
  RegisterBuilder & setWriteMaskFromContainer (const Container &container)

  Set the `writeMask` field for wide registers.

8.47.1 Detailed Description

Used to set metadata on a register resource.

8.47.2 Member Function Documentation

8.47.2.1 addEnum()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::addEnum (
            const std::string & symbol,
            const IrisValue & value,
            const std::string & description = std::string() )  [inline]
```

Add a symbol to the enums field for numeric resources.
This should be called multiple times to add multiple symbols.

Parameters

symbol	The symbol string to be associated with the specified value.
value	The value of this symbol.
description	A description of this symbol.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.2 addField()

```
FieldBuilder iris::IrisInstanceBuilder::RegisterBuilder::addField (
            const std::string & name,
            uint64_t lsbOffset,
            uint64_t bitWidth,
            const std::string & description )
```

Add a subregister field to this register. By default, the field copies attributes from its parent register, but any field can be overridden.

Parameters

name	Name of the register field.
lsbOffset	The bit offset of this field inside its parent register.
bitWidth	The size of the field.
description	Description of this field.

Returns

A FieldBuilder object that allows the caller to set attributes for this field.

8.47.2.3 addLogicalField()

```
FieldBuilder iris::IrisInstanceBuilder::RegisterBuilder::addLogicalField (
            const std::string & name,
            uint64_t bitWidth,
            const std::string & description )
```

Add a logical subregister field to this register. A logical field is a field which has a bitwidth, but which does not have an lsbOffset. It is usually used to represent non-contiguous fields which are distributed across multiple chunks in the parent register as a single contiguous register. This allows to attach enums to such a field.
By default, the field copies attributes from its parent register, but any field can be overridden.

Parameters

name	Name of the register field.
bitWidth	The size of the field.
description	Description of this field.

Returns

A FieldBuilder object that allows the caller to set attributes for this field.

8.47.2.4 addStringEnum()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::addStringEnum (
            const std::string & stringValue,
            const std::string & description = std::string() )  [inline]
```

Add a symbol to the enums field for string resources.
This should be called multiple times to add multiple symbols.

Parameters

value	The string value of this symbol. This is also used as the symbols string.
description	A description of this symbol.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.5 getRscId() [1/2]

```
ResourceId iris::IrisInstanceBuilder::RegisterBuilder::getRscId ( ) const  [inline]
```

Return the rscId that was allocated for this resource.

Returns

The rscId that was allocated for this resource.

8.47.2.6 getRscId() [2/2]

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::getRscId (
            ResourceId & rscIdOut )  [inline]
```

Get the rscId that was allocated for this resource.
This variant is useful to get the ResourceId of fields added in a chained call where return values are not practical.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.7 setAddressOffset()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setAddressOffset (
            uint64_t addressOffset )  [inline]
```

Set the `addressOffset` field.

Parameters

addressOffset	The addressOffset field of the RegisterInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.8 setBitWidth()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setBitWidth (
            uint64_t bitWidth )  [inline]
```

Set the `bitWidth` field.

Parameters

bitWidth	The bitWidth field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.9 setBreakpointSupportInfo()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setBreakpointSupportInfo (
            const std::string & supported )  [inline]
```

Set the `breakpointSupport` field.

Parameters

supported	The breakpointSupport field of the RegisterInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.10 setCanonicalRn()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setCanonicalRn (
            uint64_t canonicalRn_ )  [inline]
```

Set the `canonicalRn` field.
Note: Use setCanonicalRnElfDwarf() when using the "ElfDwarf" scheme.

Parameters

canonicalRn	The canonicalRn field of the RegisterInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.11 setCanonicalRnElfDwarf()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setCanonicalRnElfDwarf (
            uint16_t architecture,
            uint16_t dwarfRegNum )  [inline]
```

Set the `canonicalRn` field for "ElfDwarf" scheme.

Parameters

architecture	ELF EM_∗ constant for architecture.
dwarfRegNum	DWARF register number for architecture.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.12 setCname()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setCname (
```

```
            const std::string & cname )  [inline]
```

Set the `cname` field.

Parameters

cname	The cname field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.13 setDescription()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setDescription (
            const std::string & description )  [inline]
```

Set the `description` field.

Parameters

description	The description field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.14 setFormat()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setFormat (
            const std::string & format )  [inline]
```

Set the `format` field.

Parameters

format	The format field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.15 setLsbOffset()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setLsbOffset (
            uint64_t lsbOffset )  [inline]
```

Set the `lsbOffset` field.

Parameters

lsbOffset	The lsbOffset field of the RegisterInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.16 setName()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setName (
            const std::string & name )  [inline]
```

Set the `name` field.

Parameters

name	The name field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.17 setParentRscId()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setParentRscId (
            ResourceId parentRscId )  [inline]
```

Set the `parentRscId` field.
This function makes this register a child of the specified parent. It is not necessary to call this function when adding child registers using the addField() function.

Parameters

parent↩ RscId	The rscId of the parent register.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.18 setReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, ResourceReadResult &) FUNC>
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setReadDelegate ( )  [inline]
```

Set the delegate to read the resource.
Set a delegate which calls function FUNC().
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Template Parameters

FUNC	A resource read delegate function.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.19 setReadDelegate() [2/3]

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setReadDelegate (
```

```
ResourceReadDelegate readDelegate )  [inline]
```

Set the delegate to read the resource.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Parameters

readDelegate	ResourceReadDelegate object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.20 setReadDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, ResourceReadResult &) METHOD>
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setReadDelegate (
            T * instance )  [inline]
```

Set the delegate to read the resource.
Set a delegate which calls METHOD() on an instance of class T.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceReadDelegate

Template Parameters

T	A class that defines a method with the right signature to be a resource read delegate.
METHOD	A resource read delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.21 setResetData() [1/2]

```
template<typename T >
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setResetData (
            std::initializer_list< T > && t )  [inline]
```

Set the `resetData` field for wide registers.
This function accepts a braced initializer-list and is otherwise idential to setResetDataFromContainer().
Each element will be promoted/narrowed to uint64_t.

Parameters

t	Braced initializer-list.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.22 setResetData() [2/2]

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setResetData (
            uint64_t value ) [inline]
```

Set the `resetData` field to a value <= 64 bit.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

value	resetData value of the register.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.23 setResetDataFromContainer()

```
template<typename Container >
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setResetDataFromContainer (
            const Container & container ) [inline]
```

Set the `resetData` field for wide registers.
Container must be a type which allows to iterate over uint64_t bit chunks of the value,
least significant bits first, for example std::array<uint64_t> or std::vector<uint64_t>.
Each element of the container will be promoted/narrowed to uint64_t.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

container	Container containing the value in 64-bit chunks.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.24 setResetString()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setResetString (
            const std::string & resetString ) [inline]
```

Set the `resetString` field.
Set the reset value for string registers.

Parameters

resetString	The resetString field of the RegisterInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.25 setRwMode()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setRwMode (
            const std::string & rwMode ) [inline]
```

Set the `rwMode` field.

Parameters

rwMode	The rwMode field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.26 setSubRscId()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setSubRscId (
            uint64_t subRscId ) [inline]
```

Set the `subRscId` field.

Parameters

subRscId	The subRscId field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.27 setTag() [1/2]

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setTag (
            const std::string & tag ) [inline]
```

Set the named boolean tag to true (e.g. isPc)

Parameters

tag	The name of the tag to set.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.28 setTag() [2/2]

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setTag (
            const std::string & tag,
            const IrisValue & value )  [inline]
```

Set a tag to the specified value.

Parameters

tag	The name of the tag to set.
value	The value to set the tag to.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.29 setType()

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setType (
            const std::string & type )  [inline]
```

Set the `type` field.

Parameters

type	The type field of the ResourceInfo object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.30 setWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const ResourceInfo &, const ResourceWriteValue &) FUNC>
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setWriteDelegate ( )  [inline]
```

Set the delegate to write the resource.
Set a delegate which calls function FUNC().
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Template Parameters

FUNC	A resource write delegate function.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.31 setWriteDelegate() [2/3]

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setWriteDelegate (
            ResourceWriteDelegate writeDelegate )  [inline]
```

Set the delegate to write the resource.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Parameters

writeDelegate	ResourceWriteDelegate object.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.32 setWriteDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const ResourceInfo &, const ResourceWriteValue &)
METHOD>
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setWriteDelegate (
            T * instance )  [inline]
```

Set the delegate to write the resource.
Set a delegate which calls METHOD() on an instance of class T.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultResourceWriteDelegate

Template Parameters

T	A class that defines a method with the right signature to be a resource write delegate.
METHOD	A resource write delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.33 setWriteMask() [1/2]

```
template<typename T >
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setWriteMask (
            std::initializer_list< T > && t )  [inline]
```

Set the `writeMask` field for wide registers.
This function accepts a braced initializer-list and is otherwise idential to setWriteMaskFromContainer().
Each element will be promoted/narrowed to uint64_t.

Parameters

t	Braced initializer-list.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.34 setWriteMask() [2/2]

```
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setWriteMask (
            uint64_t value ) [inline]
```

Set the `writeMask` field to a value <= 64 bit.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

value	writeMask value of the register.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

8.47.2.35 setWriteMaskFromContainer()

```
template<typename Container >
RegisterBuilder & iris::IrisInstanceBuilder::RegisterBuilder::setWriteMaskFromContainer (
            const Container & container ) [inline]
```

Set the `writeMask` field for wide registers.
Container must be a type which allows to iterate over uint64_t bit chunks of the value,
least significant bits first, for example std::array<uint64_t> or std::vector<uint64_t>.
Each element of the container will be promoted/narrowed to uint64_t.
If the register is wider than the passed value the value is zero extended.
If the register is narrower than the passed value the superfluous bits are ignored.

Parameters

container	Container containing the value in 64-bit chunks.

Returns

A reference to this RegisterBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.48 iris::IrisInstanceBuilder::RegisterEventEmitterPair Struct Reference

Public Attributes

- IrisRegisterEventEmitterBase * **read**
- IrisRegisterEventEmitterBase * **update**

The documentation for this struct was generated from the following file:

- IrisInstanceBuilder.h

8.49 iris::IrisInstanceResource::ResourceInfoAndAccess Struct Reference

Entry in 'resourceInfos'.
`#include <IrisInstanceResource.h>`

Public Attributes

- ResourceReadDelegate **readDelegate**
- ResourceInfo **resourceInfo**
- ResourceWriteDelegate **writeDelegate**

8.49.1 Detailed Description

Entry in 'resourceInfos'.
Contains static resource information and information on how to access the resource.
The documentation for this struct was generated from the following file:

- IrisInstanceResource.h

8.50 iris::ResourceWriteValue Struct Reference

`#include <IrisInstanceResource.h>`

Public Attributes

- const uint64_t * **data** {}
- const std::string * **str** {}

  Non-null for non-string resources.

8.50.1 Detailed Description

Write value for ResourceWriteDelegate. This struct is used as a union. At most one of the two pointers is non-null when ResourceWriteDelegate is invoked.
The documentation for this struct was generated from the following file:

- IrisInstanceResource.h

8.51 iris::IrisInstanceBuilder::SemihostingManager Class Reference

semihosting_apis IrisInstanceBuilder semihosting APIs
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- void **enableExtensions** ()

  Instances that support semihosting extensions should call this function to enable the `IRIS_SEMIHOSTING_CALL_EXTENSION` event.
- std::vector< uint8_t > readData (uint64_t fDes, size_t max_size=0, uint64_t flags=semihost::DEFAULT)

  Read data for a given file descriptor.
- std::pair< bool, uint64_t > semihostedCall (uint64_t operation, uint64_t parameter)

Allow a client to perform a semihosting extension defined by operation and parameter.

- **SemihostingManager** (IrisInstanceSemihosting *inst_semihost_)
- void **unblock** ()
- bool **writeData** (uint64_t fDes, const std::vector< uint8_t > &data)
- bool **writeData** (uint64_t fDes, const uint8_t *data, size_t size)

8.51.1 Detailed Description

semihosting_apis IrisInstanceBuilder semihosting APIs
Manage semihosting functionality

8.51.2 Member Function Documentation

8.51.2.1 readData()

```
std::vector< uint8_t > iris::IrisInstanceBuilder::SemihostingManager::readData (
            uint64_t fDes,
            size_t max_size = 0,
            uint64_t flags = semihost::DEFAULT )  [inline]
```

Read data for a given file descriptor.
The exact behavior of this method depends on the value of the max_size and flags parameters. If the NONBLOCK flag is set, the method returns immediately with whatever data is already buffered, if any. If NONBLOCK is not set, the method blocks until data is available. Iris messages continue to be processed while this methods blocks. If max_size is not zero, then at most max_size bytes will be returned.

Parameters

fDes	File descriptor to read from. Usually semihost::STDIN.
max_size	The maximum amount of bytes to read or zero for no limit.
flags	A bitwise OR of Semihosting data request flag constants.

Returns

A vector of data that was read.

8.51.2.2 semihostedCall()

```
std::pair< bool, uint64_t > iris::IrisInstanceBuilder::SemihostingManager::semihostedCall (
            uint64_t operation,
            uint64_t parameter )  [inline]
```

Allow a client to perform a semihosting extension defined by *operation* and *parameter*.
This might implement a user-defined operation or override the default implementation for a predefined operation.

Parameters

operation	A number indicating the operation to perform. This is defined by the semihosting standard for standard operations or by the client for user-defined operations.
parameter	A parameter to the operation. The meaning of this parameter is defined by the operation.

Returns

A pair of (bool success, uint64_t result). If success is true, a client performed the function and returned the value in result. If success is false, no client performed the function and result is 0.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.52 iris::IrisInstanceMemory::SpaceInfoAndAccess Struct Reference

Entry in 'spaceInfos'.
`#include <IrisInstanceMemory.h>`

Public Attributes

- MemoryReadDelegate **readDelegate**
- MemoryGetSidebandInfoDelegate **sidebandDelegate**
- MemorySpaceInfo **spaceInfo**
- MemoryWriteDelegate **writeDelegate**

8.52.1 Detailed Description

Entry in 'spaceInfos'.
Contains static memory space information and information on how to access the space.
The documentation for this struct was generated from the following file:

- IrisInstanceMemory.h

8.53 iris::IrisInstanceBuilder::TableBuilder Class Reference

Used to set metadata for a table.
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- TableColumnBuilder addColumn (const std::string &name)

  Add a new column.
- TableBuilder & addColumnInfo (const TableColumnInfo &columnInfo)

  Add a column with a preconstructed TableColumnInfo.
- TableBuilder & setDescription (const std::string &description)

  Set the `description` field.
- TableBuilder & setFormatLong (const std::string &format)

  Set the `formatLong` field.
- TableBuilder & setFormatShort (const std::string &format)

  Set the `formatShort` field.
- TableBuilder & setIndexFormatHint (const std::string &hint)

  Set the `indexFormatHint` field.
- TableBuilder & setMaxIndex (uint64_t maxIndex)

  Set the `maxIndex` field.
- TableBuilder & setMinIndex (uint64_t minIndex)

  Set the `minIndex` field.
- TableBuilder & setName (const std::string &name)

  Set the `name` field.
- template<IrisErrorCode(∗)(const TableInfo &, uint64_t, uint64_t, TableReadResult &) FUNC>
  TableBuilder & setReadDelegate ()

  Set the delegate to read the table.
- template<typename T , IrisErrorCode(T::∗)(const TableInfo &, uint64_t, uint64_t, TableReadResult &) METHOD>
  TableBuilder & setReadDelegate (T ∗instance)

Set the delegate to read the table.

- TableBuilder & setReadDelegate (TableReadDelegate delegate)

  Set the delegate to read the table.

- template<IrisErrorCode(*)(const TableInfo &, const TableRecords &, TableWriteResult &) FUNC>
  TableBuilder & setWriteDelegate ()

  Set the delegate to write to the table.

- template<typename T , IrisErrorCode(T::*)(const TableInfo &, const TableRecords &, TableWriteResult &) METHOD>
  TableBuilder & setWriteDelegate (T *instance)

  Set the delegate to write to the table.

- TableBuilder & setWriteDelegate (TableWriteDelegate delegate)

  Set the delegate to write to the table.

- **TableBuilder** (IrisInstanceTable::TableInfoAndAccess &info_)

8.53.1 Detailed Description

Used to set metadata for a table.

8.53.2 Member Function Documentation

8.53.2.1 addColumn()

```
IrisInstanceBuilder::TableColumnBuilder iris::IrisInstanceBuilder::TableBuilder::addColumn (
            const std::string & name )  [inline]
```

Add a new column.
Call this multiple times for multiple columns

See also

AddColumnInfo

Parameters

name	The name of the new column.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.53.2.2 addColumnInfo()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::addColumnInfo (
            const TableColumnInfo & columnInfo )  [inline]
```

Add a column with a preconstructed TableColumnInfo.
Call this multiple times for multiple columns.

See also

addColumn

Parameters

columnInfo	A preconstructed TableColumnInfo object for the new column.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.3 setDescription()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setDescription (
            const std::string & description ) [inline]
```

Set the `description` field.

Parameters

description	The description field of the TableInfo object.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.4 setFormatLong()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setFormatLong (
            const std::string & format ) [inline]
```

Set the `formatLong` field.

Parameters

format	The formatLong field of the TableInfo object.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.5 setFormatShort()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setFormatShort (
            const std::string & format ) [inline]
```

Set the `formatShort` field.

Parameters

format	The formatShort field of the TableInfo object.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.6 setIndexFormatHint()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setIndexFormatHint (
            const std::string & hint ) [inline]
```

Set the `indexFormatHint` field.

Parameters

hint	The indexFormatHint field of the TableInfo object.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.7 setMaxIndex()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setMaxIndex (
            uint64_t maxIndex )  [inline]
```

Set the `maxIndex` field.

Parameters

maxIndex	The maxIndex field of the TableInfo object.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.8 setMinIndex()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setMinIndex (
            uint64_t minIndex )  [inline]
```

Set the `minIndex` field.

Parameters

minIndex	The minIndex field of the TableInfo object.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.9 setName()

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setName (
            const std::string & name )  [inline]
```

Set the `name` field.

Parameters

name	The name field of the TableInfo object.

Returns

A reference to this TableBuilder allowing calls to be chained together.

8.53.2.10 setReadDelegate() [1/3]

```
template<IrisErrorCode(*)(const TableInfo &, uint64_t, uint64_t, TableReadResult &) FUNC>
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setReadDelegate ( )  [inline]
```

Set the delegate to read the table.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultTableReadDelegate

Template Parameters

FUNC	A table read delegate function.

Returns

A reference to this TableBuilder object allowing calls to be chained together.

8.53.2.11 setReadDelegate() [2/3]

```
template<typename T , IrisErrorCode(T::*)(const TableInfo &, uint64_t, uint64_t, TableRead↩
Result &) METHOD>
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setReadDelegate (
            T * instance )  [inline]
```

Set the delegate to read the table.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultTableReadDelegate

Template Parameters

T	A class that defines a method with the right signature to be a table read delegate.
METHOD	A table read delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this TableBuilder object allowing calls to be chained together.

8.53.2.12 setReadDelegate() [3/3]

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setReadDelegate (
            TableReadDelegate delegate )  [inline]
```

Set the delegate to read the table.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultTableReadDelegate

Parameters

delegate	TableReadDelegate object.

Returns

A reference to this TableBuilder object allowing calls to be chained together.

8.53.2.13 setWriteDelegate() [1/3]

```
template<IrisErrorCode(*)(const TableInfo &, const TableRecords &, TableWriteResult &) FUNC>
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setWriteDelegate ( )  [inline]
```

Set the delegate to write to the table.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultTableWriteDelegate

Template Parameters

FUNC	A table write delegate function.

Returns

A reference to this TableBuilder object allowing calls to be chained together.

8.53.2.14 setWriteDelegate() [2/3]

```
template<typename T , IrisErrorCode(T::*)(const TableInfo &, const TableRecords &, Table↩
WriteResult &) METHOD>
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setWriteDelegate (
            T * instance )  [inline]
```

Set the delegate to write to the table.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultTableWriteDelegate

Template Parameters

T	A class that defines a method with the right signature to be a table write delegate.
METHOD	A table write delegate method in class T.

Parameters

instance	The instance of class T on which to call METHOD.

Returns

A reference to this TableBuilder object allowing calls to be chained together.

8.53.2.15 setWriteDelegate() [3/3]

```
TableBuilder & iris::IrisInstanceBuilder::TableBuilder::setWriteDelegate (
            TableWriteDelegate delegate )  [inline]
```

Set the delegate to write to the table.
If this is not set, the default delegate is used.

See also

IrisInstanceBuilder::setDefaultTableWriteDelegate

Parameters

delegate	TableWriteDelegate object.

Returns

A reference to this TableBuilder object allowing calls to be chained together.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.54 iris::IrisInstanceBuilder::TableColumnBuilder Class Reference

Used to set metadata for a table column.
`#include <IrisInstanceBuilder.h>`

Public Member Functions

- TableColumnBuilder addColumn (const std::string &name)
  Add another new column.
- TableBuilder & addColumnInfo (const TableColumnInfo &columnInfo)
  Add another column with a preconstructed TableColumnInfo.
- TableBuilder & endColumn ()
  Stop building this column and go back to the parent table.
- TableColumnBuilder & setBitWidth (uint64_t bitWidth)
  Set the `bitWidth` field.
- TableColumnBuilder & setDescription (const std::string &description)
  Set the `description` field.
- TableColumnBuilder & setFormat (const std::string &format)
  Set the `format` field.
- TableColumnBuilder & setFormatLong (const std::string &format)
  Set the `formatLong` field.
- TableColumnBuilder & setFormatShort (const std::string &format)
  Set the `formatShort` field.

- TableColumnBuilder & setName (const std::string &name)

  Set the `name` field.
- TableColumnBuilder & setRwMode (const std::string &rwMode)

  Set the `rwMode` field.
- TableColumnBuilder & setType (const std::string &type)

  Set the `type` field.
- **TableColumnBuilder** (TableBuilder &parent_, TableColumnInfo &info_)

8.54.1 Detailed Description

Used to set metadata for a table column.

8.54.2 Member Function Documentation

8.54.2.1 addColumn()

```
TableColumnBuilder iris::IrisInstanceBuilder::TableColumnBuilder::addColumn (
            const std::string & name )  [inline]
```

Add another new column.
Call this multiple times for multiple columns

See also

TableBuilder::addColumn

Parameters

name	The name of the new column.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.2 addColumnInfo()

```
TableBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::addColumnInfo (
            const TableColumnInfo & columnInfo )  [inline]
```

Add another column with a preconstructed TableColumnInfo.

See also

TableBuilder::addColumnInfo

addColumn

Parameters

columnInfo	A preconstructed TableColumnInfo object for the new column.

Returns

A reference to the parent TableBuilder for this table.

8.54.2.3 endColumn()

```
TableBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::endColumn ( )  [inline]
```

Stop building this column and go back to the parent table.

See also

addColumn

addColumnInfo

Returns

The parent TableBuilder for this table.

8.54.2.4 setBitWidth()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setBitWidth (
            uint64_t bitWidth )  [inline]
```

Set the `bitWidth` field.

Parameters

bitWidth	The bitWidth field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.5 setDescription()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setDescription (
            const std::string & description )  [inline]
```

Set the `description` field.

Parameters

description	The description field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.6 setFormat()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setFormat (
            const std::string & format )  [inline]
```

Set the `format` field.

Parameters

format	The format field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.7 setFormatLong()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setFormatLong (
            const std::string & format )  [inline]
```

Set the `formatLong` field.

Parameters

format	The formatLong field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.8 setFormatShort()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setFormatShort (
            const std::string & format )  [inline]
```

Set the `formatShort` field.

Parameters

format	The formatShort field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.9 setName()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setName (
            const std::string & name )  [inline]
```

Set the `name` field.

Parameters

name	The name field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.10 setRwMode()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setRwMode (
            const std::string & rwMode )  [inline]
```

Set the `rwMode` field.

Parameters

rwMode	The rwMode field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

8.54.2.11 setType()

```
TableColumnBuilder & iris::IrisInstanceBuilder::TableColumnBuilder::setType (
            const std::string & type )  [inline]
```

Set the `type` field.

Parameters

type	The type field of the TableColumnInfo object.

Returns

A TableColumnBuilder object that can be used to add metadata for the new column.

The documentation for this class was generated from the following file:

- IrisInstanceBuilder.h

8.55 iris::IrisInstanceTable::TableInfoAndAccess Struct Reference

Entry in 'tableInfos'.
`#include <IrisInstanceTable.h>`

Public Attributes

- TableReadDelegate **readDelegate**

  Can be empty, in which case defaultReadDelegate is used.
- TableInfo **tableInfo**
- TableWriteDelegate **writeDelegate**

  Can be empty, in which case defaultWriteDelegate is used.

8.55.1 Detailed Description

Entry in 'tableInfos'.
Contains static table information and information on how to access the table.
The documentation for this struct was generated from the following file:

- IrisInstanceTable.h

Chapter 9

File Documentation

9.1 IrisCanonicalMsnArm.h File Reference

Constants for the memory.canonicalMsnScheme arm.com/memoryspaces.

```
#include "iris/detail/IrisInterface.h"
#include "iris/detail/IrisCommon.h"
```

Enumerations

- enum **CanonicalMsnArm** : uint64_t {
  CanonicalMsnArm_SecureMonitor = 0x1000 , **CanonicalMsnArm_Secure** = 0x1000 , **CanonicalMsn↩ Arm_Guest** = 0x1001 , **CanonicalMsnArm_Normal** = 0x1001 ,
  CanonicalMsnArm_NSHyp = 0x1002 , **CanonicalMsnArm_Memory** = 0x1003 , **CanonicalMsnArm_↩ HypApp** = 0x1004 , **CanonicalMsnArm_Host** = 0x1005 ,
  CanonicalMsnArm_Current = 0x10ff , **CanonicalMsnArm_IPA** = 0x1100 , **CanonicalMsnArm_Physical↩ MemorySecure** = 0x1200 , **CanonicalMsnArm_PhysicalMemoryNonSecure** = 0x1201 ,
  CanonicalMsnArm_PhysicalMemory = 0x1202 , **CanonicalMsnArm_PhysicalMemoryRoot** = 0x1203 ,
  CanonicalMsnArm_PhysicalMemoryRealm = 0x1204 }

9.1.1 Detailed Description

Constants for the memory.canonicalMsnScheme arm.com/memoryspaces.

Date

9.2 IrisCanonicalMsnArm.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisCanonicalMsnArm_h
9 #define ARM_INCLUDE_IrisCanonicalMsnArm_h
10
11 #include "iris/detail/IrisInterface.h" // uint64_t
12 #include "iris/detail/IrisCommon.h"    // namespace iris
13
14 NAMESPACE_IRIS_START
15
16 enum CanonicalMsnArm: uint64_t
17 {
18     CanonicalMsnArm_SecureMonitor = 0x1000,    CanonicalMsnArm_Secure        = 0x1000,
19     CanonicalMsnArm_Guest         = 0x1001,    CanonicalMsnArm_Normal        = 0x1001,
20     CanonicalMsnArm_NSHyp         = 0x1002,
21     CanonicalMsnArm_Memory        = 0x1003, // Virtual memory for cores which do not have TrustZone.
22     CanonicalMsnArm_HypApp        = 0x1004,
23     CanonicalMsnArm_Host          = 0x1005,
24
25     CanonicalMsnArm_Current       = 0x10ff,
26
```

```
27     CanonicalMsnArm_IPA           = 0x1100,
28
29     CanonicalMsnArm_PhysicalMemorySecure     = 0x1200,
30     CanonicalMsnArm_PhysicalMemoryNonSecure  = 0x1201,
31     CanonicalMsnArm_PhysicalMemory           = 0x1202,
32     CanonicalMsnArm_PhysicalMemoryRoot       = 0x1203,
33     CanonicalMsnArm_PhysicalMemoryRealm      = 0x1204
34 }; // enum CanonicalMsnArm
35
36 NAMESPACE_IRIS_END
37
38 #endif // ARM_INCLUDE_IrisCanonicalMsnArm_h
39
```

9.3 IrisCConnection.h File Reference

IrisConnectionInterface implementation based on IrisC.

```
#include "iris/detail/IrisC.h"
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisErrorException.h"
#include "iris/detail/IrisInterface.h"
#include "iris/detail/IrisJsonProducer.h"
#include <string>
```

Classes

- class iris::IrisCConnection

  Provide an IrisConnectionInterface which loads an IrisC library.

9.3.1 Detailed Description

IrisConnectionInterface implementation based on IrisC.

Copyright

Copyright (C) 2017-2024 Arm Limited. All rights reserved.

9.4 IrisCConnection.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisCConnection_h
8 #define ARM_INCLUDE_IrisCConnection_h
9
10 #include "iris/detail/IrisC.h"
11 #include "iris/detail/IrisCommon.h"
12 #include "iris/detail/IrisErrorException.h"
13 #include "iris/detail/IrisInterface.h"
14 #include "iris/detail/IrisJsonProducer.h"
15
16 #include <string>
17
18 NAMESPACE_IRIS_START
19
25 class IrisCConnection : public IrisConnectionInterface
26 {
27 private:
29     IrisC_HandleMessageFunction   handleMessage_function;
30
31     IrisC_RegisterChannelFunction   registerChannel_function;
32     IrisC_UnregisterChannelFunction unregisterChannel_function;
33
34     IrisC_ProcessAsyncMessagesFunction processAsyncMessages_function;
35
36     class RemoteInterface : public IrisInterface
37     {
38     private:
39         IrisCConnection* irisc;
40
41     public:
42         RemoteInterface(IrisCConnection* irisc_)
```

```
43              : irisc(irisc_)
44          {
45          }
46
47      public: // IrisInterface
48          virtual void irisHandleMessage(const uint64_t* message) override
49          {
50              // Forward to the IrisC library
51              int64_t status = irisc->IrisC_handleMessage(message);
52
53              if (status != E_ok)
54              {
55                  throw IrisErrorException(IrisErrorCode(status));
56              }
57          }
58      } remote_interface;
59
60      // Helper function to bridge IrisC_HandleMessageFunction to IrisInterface::irisHandleMessage
61      static int64_t handleMessageToIrisInterface(void* context, const uint64_t* message)
62      {
63          if (context == nullptr)
64          {
65              return E_invalid_context;
66          }
67          try
68          {
69              static_cast<IrisInterface*>(context)->irisHandleMessage(message);
70          }
71          catch (std::exception& e)
72          {
73              // Catch and print all exceptions here as they usually get silently dropped when going
74              // back through the C function.
75              // These are always programming errors (e.g. in plugin event callbacks) and not
76              // valid error return values of Iris functions.
77              std::cout « "Caught exception on plugin C boundary: " « e.what() « "\n";
78              std::cout « "Call was: " « messageToString(message) «  "\n";
79
80              // Some compilers can transport exceptions through C functions, some not.
81              // Do whatever the compiler can do.
82              throw;
83          }
84
85          return E_ok;
86      }
87
88  protected:
90      void* iris_c_context;
91
93      IrisCConnection()
94          : handleMessage_function(nullptr)
95          , registerChannel_function(nullptr)
96          , unregisterChannel_function(nullptr)
97          , processAsyncMessages_function(nullptr)
98          , remote_interface(this)
99          , iris_c_context(nullptr)
100     {
101     }
102
104
105     int64_t IrisC_handleMessage(const uint64_t* message)
106     {
107         return (*handleMessage_function)(iris_c_context, message);
108     }
109
110     int64_t IrisC_registerChannel(IrisC_CommunicationChannel* channel, uint64_t* channel_id_out)
111     {
112         return (*registerChannel_function)(iris_c_context, channel, channel_id_out);
113     }
114
115     int64_t IrisC_unregisterChannel(uint64_t channel_id)
116     {
117         return (*unregisterChannel_function)(iris_c_context, channel_id);
118     }
119
120     int64_t IrisC_processAsyncMessages(bool waitForAMessage)
121     {
122         return (*processAsyncMessages_function)(iris_c_context, waitForAMessage);
123     }
124
125 public:
126     IrisCConnection(IrisC_Functions* functions)
127         : handleMessage_function(functions->handleMessage_function)
128         , registerChannel_function(functions->registerChannel_function)
129         , unregisterChannel_function(functions->unregisterChannel_function)
130         , processAsyncMessages_function(functions->processAsyncMessages_function)
131         , remote_interface(this)
132         , iris_c_context(functions->iris_c_context)
```

```
133     {
134     }
135
136 public: // IrisConnectionInterface
141     virtual uint64_t registerIrisInterfaceChannel(IrisInterface* iris_interface, const std::string&
        connectionInfo) override
142     {
143         (void)connectionInfo;
144         IrisC_CommunicationChannel channel;
145
146         channel.CommunicationChannel_version = 0;
147         channel.handleMessage_function       = &IrisCConnection::handleMessageToIrisInterface;
148         channel.handleMessage_context        = static_cast<void*>(iris_interface);
149
150         uint64_t channelId = IRIS_UINT64_MAX;
151
152         IrisErrorCode status = static_cast<IrisErrorCode>(IrisC_registerChannel(&channel, &channelId));
153
154         if (status != E_ok)
155         {
156             throw IrisErrorException(status);
157         }
158
159         return channelId;
160     }
161
166     virtual void unregisterIrisInterfaceChannel(uint64_t channelId) override
167     {
168         IrisErrorCode status = static_cast<IrisErrorCode>(IrisC_unregisterChannel(channelId));
169
170         if (status != E_ok)
171         {
172             throw IrisErrorException(status);
173         }
174     }
175
180     virtual IrisErrorCode processAsyncMessages(bool waitForAMessage) override
181     {
182         return static_cast<IrisErrorCode>(IrisC_processAsyncMessages(waitForAMessage));
183     }
184
189     virtual IrisInterface* getIrisInterface() override
190     {
191         return &remote_interface;
192     }
193 };
194
195 NAMESPACE_IRIS_END
196
197 #endif // ARM_INCLUDE_IrisCConnection_h
```

9.5 IrisClient.h File Reference

Iris client which supports multiple methods to connect to other Iris executables.

```
#include "iris/IrisInstance.h"
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisErrorCode.h"
#include "iris/detail/IrisInterface.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisUtils.h"
#include "iris/detail/IrisCommaSeparatedParameters.h"
#include "iris/impl/IrisChannelRegistry.h"
#include "iris/impl/IrisMessageQueue.h"
#include "iris/impl/IrisPlugin.h"
#include "iris/impl/IrisProcessEventsThread.h"
#include "iris/impl/IrisRpcAdapterTcp.h"
#include "iris/impl/IrisTcpSocket.h"
#include "iris/impl/IrisSocketpairPool.h"
#include <map>
#include <list>
#include <memory>
#include <mutex>
#include <queue>
#include <thread>
```

```
#include <vector>
```

Classes

- class iris::IrisClient

Functions

- **NAMESPACE_IRIS_INTERNAL_START** (service) class IrisServiceTcpServer

9.5.1 Detailed Description

Iris client which supports multiple methods to connect to other Iris executables.

Date

Copyright ARM Limited 2015-2024 All Rights Reserved.

9.6 IrisClient.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisClient_h
8 #define ARM_INCLUDE_IrisClient_h
9
10 #include "iris/IrisInstance.h"
11
12 #include "iris/detail/IrisCommon.h"
13 #include "iris/detail/IrisErrorCode.h"
14 #include "iris/detail/IrisInterface.h"
15 #include "iris/detail/IrisLogger.h"
16 #include "iris/detail/IrisUtils.h"
17 #include "iris/detail/IrisCommaSeparatedParameters.h"
18
19 #include "iris/impl/IrisChannelRegistry.h"
20 #include "iris/impl/IrisMessageQueue.h"
21 #include "iris/impl/IrisPlugin.h"
22 #include "iris/impl/IrisProcessEventsThread.h"
23 #include "iris/impl/IrisRpcAdapterTcp.h"
24 #include "iris/impl/IrisTcpSocket.h"
25 #include "iris/impl/IrisSocketpairPool.h"
26 #include "iris/IrisInstance.h"
27
28 #include <map>
29 #include <list>
30 #include <memory>
31 #include <mutex>
32 #include <queue>
33 #include <thread>
34 #include <vector>
35 #if defined(__linux__) || defined(__APPLE__)
36 #include <csignal>
37 #include <sys/types.h>
38 #include <sys/wait.h>
39 #endif
40 #if defined(__linux__)
41 #include <sys/prctl.h>
42 #endif
43
44 NAMESPACE_IRIS_INTERNAL_START(service)
45 class IrisServiceTcpServer;
46 NAMESPACE_IRIS_INTERNAL_END
47
48 NAMESPACE_IRIS_START
49
50 class IrisClient
51     : public IrisInterface
52     , public impl::IrisProcessEventsInterface
53     , public IrisConnectionInterface
54     , public IrisInstance
55 {
56 public:
59     IrisClient(const std::string& instName, const std::vector<std::string>& commandLine_, const
       std::string& programName)
60     {
```

```
61          // try/catch: Always destroy this object when an exception is thrown in this convenience
        constructor. We consider ourselves always fully initialized.
62          try
63          {
64              init(IRIS_TCP_CLIENT, instName);
65              connectCommandLine(commandLine_, programName);
66          }
67          catch (...)
68          {
69              destructor();
70              throw;
71          }
72      }
73
75      IrisClient(const std::string& instName = std::string(), const std::string& connectionSpec =
        std::string())
76      {
77          // try/catch: Always destroy this object when an exception is thrown in this convenience
        constructor. We consider ourselves always fully initialized.
78          try
79          {
80              init(IRIS_TCP_CLIENT, instName);
81              if (!connectionSpec.empty())
82              {
83                  connect(connectionSpec);
84              }
85          }
86          catch (...)
87          {
88              destructor();
89              throw;
90          }
91      }
92
94      IrisClient(const service::IrisServiceTcpServer*, const std::string& instName = std::string())
95      {
96          init(IRIS_SERVICE_SERVER, instName);
97      }
98
109      IrisClient(const std::string& hostname, uint16_t port, const std::string& instName = std::string())
110      {
111          // try/catch: Always destroy this object when an exception is thrown in this convenience
        constructor. We consider ourselves always fully initialized.
112          try
113          {
114              init(IRIS_TCP_CLIENT, instName);
115              std::string ignored_error;
116              IrisErrorCode status = connect(hostname, port, port ? 1000 : 100, ignored_error);
117              if (status != E_ok)
118              {
119                  throw IrisErrorExceptionString(status, "Failed to connect to Iris TCP server");
120              }
121          }
122          catch (...)
123          {
124              destructor();
125              throw;
126          }
127      }
128
130      virtual ~IrisClient()
131      {
132          destructor();
133      }
134
148      void connectCommandLine(const std::vector<std::string>& commandLine_, const std::string&
        programName)
149      {
150          std::vector<std::string> commandLine = commandLine_;
151          connectCommandLineInternal(commandLine, programName, /*keepOtherArgs*/false);
152      }
153
155      void connectCommandLineKeepOtherArgs(std::vector<std::string>& commandLine, const std::string&
        programName)
156      {
157          connectCommandLineInternal(commandLine, programName, /*keepOtherArgs*/true);
158      }
159
162      static std::string getConnectCommandLineHelp()
163      {
164          return
165              "Iris connection options:\n"
166              "  Start a model child process and connect to it using UNIX domain sockets:\n"
167              "    %prog [OPTIONS] [timeout=TIMEOUT_IN_MS] [iris-log[=N]] [verbose[=0..3]]
        [server_verbose[=0..3]] -- MODEL [MODEL_OPTIONS...]\n"
168              "\n"
169              "  Connect to an already running model process using TCP:\n"
```

```
170             "    %prog [OPTIONS] [tcp[=HOST]] [port=PORT] [timeout=TIMEOUT_IN_MS] [iris-log[=N]]
    [verbose[=0..3]]\n"
171             "\n"
172             "  The arguments have the following semantics:\n"
173             "    --: Start a model child process and connect to it using UNIX domain sockets. The model
    command line follows after '--'.\n"
174             "    tcp[=HOST]: (tcp only) Use TCP to connect to model process. Set hostname. Default is
    localhost.\n"
175             "    port=N: (tcp only) Set Iris server port. Default is 0 if HOST is localhost, else 7100.
    0 means scan ports 7100..7109.\n"
176             "    timeout=N: Set connection timeout to N ms. Default is 100 ms for \"tcp\" if PORT is 0,
    else 1000 ms. Set this to 60000 ms when starting under gdb with \"set follow-fork-mode child\".\n"
177             "    iris-log[=N]: Log Iris functions calls (1=pretty, 2=JSON, 3=JSON-multiline, +8=U64JSON,
    +16=time, +32=reltime).\n"
178             "    verbose=N: Set verbose level of IrisClient (0..3).\n"
179             "    server_verbose: (-- only) Set verbose level of Iris server (0..3). (For \"tcp\" set
    server verbose level on model command line when starting the model.)\n"
180             "    help: Print this connection option help message.\n"
181             "\n"
182             "  Example: Start model and connect to it:\n"
183             "    %prog -- isim_system -C bp.secure_memory=false -a cluster0.cpu0=hello.axf\n"
184             "  Example: Same but also log Iris function calls and increase connection timeout to 60s
    (useful when debugging model under gdb with \"set follow-fork-mode child\"):\n"
185             "    %prog iris-log timeout=60000 -- isim_system -C bp.secure_memory=false -a
    cluster0.cpu0=hello.axf\n"
186             "  Example: Connect to first model process found while scanning ports 7100..7109 on
    localhost:\n"
187             "    %prog\n"
188             "  Example: Connect to model process on host 10.10.10.10 and port 7101:\n"
189             "    %prog tcp=10.10.10.10 port=7101\n"
190             "\n"
191             ;
192     }
193
198     void spawnAndConnect(const std::vector<std::string>& modelCommandLine, const std::string&
    additionalServerArgs = std::string(), const std::string& additionalClientArgs = std::string())
199     {
200 #ifdef _WIN32
201         (void)modelCommandLine;
202         (void)additionalServerArgs;
203         (void)additionalClientArgs;
204         if (modelCommandLine.size() < 1000000) // Hack: Disable spurious "unreachable code" warning in
    code calling spawnAndConnect() on Windows while we have not implemented this.
205         {
206             throw IrisErrorExceptionString(E_not_connected, "socketpair() connections not yet supported
    on Windows");
207         }
208 #else
209         // Increase verbose level? (connect() below does this, but is too late)
210         IrisCommaSeparatedParameters clientArgs(additionalClientArgs, "1");
211         setVerbose(unsigned(clientArgs.getUint("verbose", 0)), /*increaseOnly=*/true);
212         setIrisMessageLogLevel(unsigned(clientArgs.getUint("iris-log", 0)), /*increaseOnly=*/true);
213         if (verbose)
214         {
215             log.info("IrisClient::spawnAndConnect(modelCommandLine=" + toStr(modelCommandLine) + ",
    additionalServerArgs=" + quoteStringToJson(additionalServerArgs) + ", additionalClientArgs=" +
    quoteStringToJson(additionalClientArgs) + ")\n");
216         }
217
218         if (isConnected() || (childPid > 0))
219         {
220             disconnectAndWaitForChildToExit();
221         }
222
223         // Create socketpair pool.
224         if (!socketpairPool)
225         {
226             socketpairPoolToDelete = socketpairPool = new SocketpairPool(2);
227         }
228
229         // Get socket pair.
230         Socketpair socketPair = socketpairPool->allocSocketpair();
231
232         lastExitStatus = -1;
233
234         // Fork.
235         childPid = uint64_t(::fork());
236         if (childPid == 0)
237         {
238             // Child == server/model.
239             close(socketPair.clientFd);
240
241 #if defined(__linux__)
242             // Ask the kernel to kill us with SIGINT on parent thread termination.
243             // NOTE: Cleared on fork, but not on exec.
244             prctl(PR_SET_PDEATHSIG, SIGINT);
245 #endif
```

```
246
247             // Prepare args.
248             std::vector<std::string> args = modelCommandLine;
249             args.push_back("--iris-connect");
250             args.push_back("socketfd=" + std::to_string(socketPair.serverFd) + "," +
    additionalServerArgs);
251             std::vector<const char *> cargs;
252             for (const std::string& s: args)
253             {
254                 cargs.push_back(s.c_str());
255             }
256             cargs.push_back(nullptr);
257
258             // Start model. Replaces the currently running executable. Does not return on success.
259             execve(cargs[0], (char * const *)cargs.data(), environ);
260
261             // execve() only returns on error.
262             // When we get here the model could not be started.
263
264             // Send error message to client which will print it.
265             std::stringstream os;
266             os « "IrisRpc/1.0 400 Starting model command line failed: " « strerror(errno) « ": " «
    iris::joinString(args, " ") « ".\r\n\r\n";
267             std::string s = os.str();
268             auto error = ::write(socketPair.serverFd, s.data(), s.length());
269             (void)error;
270             close(socketPair.serverFd);
271
272             // The execve() failed and we lost all other threads after the fork().
273             // We cannot exit through an error exception and through main() since all thread destructors
    will hang.
274             // Exit here.
275             ::exit(1);
276         }
277         else if (int64_t(childPid) < 0)
278         {
279             close(socketPair.clientFd);
280             close(socketPair.serverFd);
281             childPid = 0;
282             throw IrisErrorExceptionString(E_not_connected, "fork() failed with errno=" +
    std::to_string(errno) + ".");
283         }
284         else
285         {
286             if (verbose)
287             {
288                 log.info("IrisClient::spawnAndConnect(): Spawned child process %d.\n", int(childPid));
289             }
290
291             // Parent == client/debugger.
292             close(socketPair.serverFd);
293
294             try
295             {
296                 // Connect to model.
297                 connect("socketfd=" + std::to_string(socketPair.clientFd) + "," + additionalClientArgs);
298             }
299             catch (...)
300             {
301                 // connect() already closed the socket on error.
302
303                 // Issue SIGINT and then SIGKILL to terminate child.
304                 disconnectAndWaitForChildToExit(0);
305                 throw;
306             }
307         }
308 #endif
309     }
310
324     bool disconnectAndWaitForChildToExit(double timeoutInMs = 5000, double timeoutInMsAfterSigInt =
    5000, double timeoutInMsAfterSigKill = 5000)
325     {
326         if (verbose)
327         {
328             log.info("IrisClient::disconnectAndWaitForChildToExit(timeoutInMs=%.0f,
    timeoutInMsAfterSigInt=%.0f, timeoutInMsAfterSigKill=%.0f)\n", timeoutInMs, timeoutInMsAfterSigInt,
    timeoutInMsAfterSigKill);
329         }
330
331         // Disconnect.
332         IrisErrorCode error = disconnect();
333         if (error)
334         {
335             throw IrisErrorExceptionString(E_not_connected, "disconnect() failed.");
336         }
337
338 #ifdef _WIN32
```

```
339         (void)timeoutInMs;
340         (void)timeoutInMsAfterSigInt;
341         (void)timeoutInMsAfterSigKill;
342         throw IrisErrorExceptionString(E_not_implemented, "socketpair() connections not yet supported on
    Windows.");
343 #else
344         if (childPid == 0)
345         {
346             return true;
347         }
348 
349         if (!floatEqual(timeoutInMs, 0.0))
350         {
351             // Wait for child process to exit for timeoutInMs.
352             if (waitpidWithTimeout(childPid, &lastExitStatus, 0, timeoutInMs))
353             {
354                 childPid = 0;
355                 return true;
356             }
357         }
358 
359         if (!floatEqual(timeoutInMsAfterSigInt, 0.0))
360         {
361             // Send SIGINT and wait for timeoutInMsAfterSigInt.
362             if (verbose)
363             {
364                 log.info("IrisClient::disconnectAndWaitForChildToExit(): Sending SIGINT to child %d.\n",
    int(childPid));
365             }
366             if (kill(pid_t(childPid), SIGINT) < 0)
367             {
368                 throw IrisErrorExceptionString(E_not_connected, "kill(SIGINT) failed with errno=" +
    std::to_string(errno) + ".");
369             }
370             if (waitpidWithTimeout(childPid, &lastExitStatus, 0, timeoutInMsAfterSigInt))
371             {
372                 childPid = 0;
373                 return true;
374             }
375         }
376 
377         if (!floatEqual(timeoutInMsAfterSigKill, 0.0))
378         {
379             // Send SIGKILL and wait for timeoutInMsAfterSigKill.
380             if (verbose)
381             {
382                 log.info("IrisClient::disconnectAndWaitForChildToExit(): Sending SIGKILL to child
    %d.\n", int(childPid));
383             }
384             if (kill(pid_t(childPid), SIGKILL) < 0)
385             {
386                 throw IrisErrorExceptionString(E_not_connected, "kill(SIGKILL) failed with errno=" +
    std::to_string(errno) + ".");
387             }
388             if (waitpidWithTimeout(childPid, &lastExitStatus, 0, timeoutInMsAfterSigKill))
389             {
390                 childPid = 0;
391                 return true;
392             }
393         }
394 
395         // Child did not exit so far.
396         if (verbose)
397         {
398             log.info("IrisClient::disconnectAndWaitForChildToExit(): Child %d did not exit.\n",
    int(childPid));
399         }
400         return false;
401 #endif
402     }
403 
404 #ifndef _WIN32
408     bool waitpidWithTimeout(uint64_t pid, int* status, int options, double timeoutInMs)
409     {
410         if (verbose)
411         {
412             log.info("IrisClient::waitpidWithTimeout(): Waiting %.1f ms for child %d to exit ...\n",
    timeoutInMs, int(pid));
413         }
414 
415         double endTime = getTimeInSec() + timeoutInMs / 1000.0;
416         if (timeoutInMs < 0)
417         {
418             endTime += 1e100;
419         }
420 
421         // Wait for child to exit.
```

```
422         while (getTimeInSec() < endTime)
423         {
424             pid_t ret = waitpid(pid_t(pid), status, options | WNOHANG);
425             if (ret == pid_t(pid))
426             {
427                 if (verbose)
428                 {
429                     log.info("IrisClient::waitpidWithTimeout(): Child %d exited with exit status %d
    after waiting for %.3fs.\n", int(pid), status ? *status : 0, getTimeInSec() - endTime + (timeoutInMs
    / 1000.0));
430                 }
431                 return true; // Child exited.
432             }
433             if (ret < 0)
434             {
435                 throw IrisErrorExceptionString(E_not_connected, "waitpid() failed with errno=" +
    std::to_string(errno) + ".");
436             }
437             if (ret > 0)
438             {
439                 throw IrisErrorExceptionString(E_not_connected, "waitpid() returned unexpected pid=" +
    std::to_string(pid) + ".");
440             }
441             assert(ret == 0);
442
443             sleepMs(20);
444         }
445
446         return false; // Timeout.
447     }
448 #endif
449
451     uint64_t getChildPid() const
452     {
453         return childPid;
454     }
455
457     int getLastExitStatus() const { return lastExitStatus; }
458
460     const std::string connectionHelpStr =
461         "Supported connection types:\n"
462         "tcp[=HOST][,port=PORT][,timeout=T]\n"
463         "  Connect to an Iris TCP server on HOST:PORT.\n"
464         "  The default for HOST is 'localhost' and the default for PORT is 0 if HOST is 'localhost' and
    7100 otherwise. If PORT is 0 then a port scan on ports 7100 to 7109 is done.\n"
465         "  T is the connection timeout in ms (defaults to 100 if PORT==0, else 1000).\n"
466         "\n"
467         "socketfd=FD[,timeout=T]\n"
468         "  Use socket file descriptor FD as an established UNIX domain socket connection.\n"
469         "  T is the timeout for the Iris handshake in ms.\n"
470         "\n"
471         "General parameters:\n"
472         "  verbose[=N]: Increase verbose level of IrisClient to level N (0..3).\n"
473         "  iris-log[=N]: Log Iris functions calls (1=pretty, 2=JSON, 3=JSON-multiline, +8=U64JSON,
    +16=time, +32=reltime).\n";
474
483     void connect(const std::string& connectionSpec)
484     {
485         IrisCommaSeparatedParameters params(connectionSpec, "1");
486
487         // Emit help message?
488         if (params.have("help"))
489         {
490             throw IrisErrorExceptionString(E_help_message, connectionHelpStr);
491         }
492
493         // Increase verbose level?
494         setVerbose(unsigned(params.getUint("verbose", 0)), /*increaseOnly=*/true);
495         setIrisMessageLogLevel(unsigned(params.getUint("iris-log", 0)), /*increaseOnly=*/true);
496         if (verbose)
497         {
498             log.info("IrisClient::connect(connectionSpec=" + quoteStringToJson(connectionSpec) + ")\n");
499         }
500
501         // Validate connection type.
502         if (unsigned(params.have("tcp")) + unsigned(params.have("socketfd")) != 1)
503         {
504             throw IrisErrorExceptionString(E_not_connected, "Exactly one out of \"tcp\", \"socketfd\"
    and \"help\" must be specified (got \"" + connectionSpec + "\"). Specify \"help\" to get a list of
    all supported connection types.");
505         }
506
507         if (params.have("tcp"))
508         {
509             std::string hostname = params.getStr("tcp");
510             if (hostname == "1")
511             {
```

```
512                 hostname = "localhost";
513             }
514             uint16_t port = uint16_t(params.getUint("port", hostname == "localhost" ? 0 : 7100));
515             unsigned timeoutInMs = unsigned(params.getUint("timeout", port == 0 ? 100 : 1000));
516             if (params.haveUnusedParameters())
517             {
518                 throw IrisErrorExceptionString(E_not_connected, params.getUnusedParametersMessage("Error
    in 'tcp' connection parameters: "));
519             }
520             std::string errorResponse;
521             IrisErrorCode status = connect(hostname, port, timeoutInMs, errorResponse);
522             if (status != E_ok)
523             {
524                 throw IrisErrorExceptionString(status, errorResponse);
525             }
526         }
527 
528         if (params.have("socketfd"))
529         {
530             SocketFd socketfd = SocketFd(params.getUint("socketfd"));
531             unsigned timeoutInMs = unsigned(params.getUint("timeout", 1000));
532             if (params.haveUnusedParameters())
533             {
534                 throw IrisErrorExceptionString(E_not_connected, params.getUnusedParametersMessage("Error
    in 'socketfd' connection parameters: "));
535             }
536             connectSocketFd(socketfd, timeoutInMs);
537         }
538     }
539 
544     IrisErrorCode connect(const std::string& hostname, uint16_t port, unsigned timeoutInMs, std::string&
    errorResponseOut)
545     {
546         assert(mode == IRIS_TCP_CLIENT);
547 
548         if (verbose)
549             log.info("IrisClient::connect(hostname=%s, port=%u, timeout=%u) enter\n", hostname.c_str(),
    port, timeoutInMs);
550 
551         // Already connected?
552         IrisErrorCode error = E_ok;
553         if (adapter.isConnected() || sock.isConnected())
554         {
555             error = E_already_connected;
556             goto done;
557         }
558 
559         // hostname==localhost and port==0 means port scan.
560         if ((hostname == "localhost") && (port == 0))
561         {
562             const uint16_t startport = 7100;
563             const uint16_t endport = 7109;
564             for (port = startport; port <= endport; port++)
565             {
566                 std::string errorMessage;
567                 if (connect(hostname, port, timeoutInMs, errorResponseOut) == iris::E_ok)
568                     return E_ok;
569             }
570             errorResponseOut = "No Iris TCP server found on ports " + std::to_string(startport) + ".." +
    std::to_string(endport) + "\n";
571             error = E_not_connected;
572             goto done;
573         }
574 
575         if (!sock.isCreated())
576         {
577             sock.create();
578             sock.setNonBlocking();
579 
580             // Unblock a potentially blocked worker thread which so far is waiting indefinitely
581             // on 'no socket'. This thread will block again on the socket we just created.
582             socketSet.stopWaitForEvent();
583         }
584 
585         // Connect to server.
586         error = sock.connect(hostname, port, timeoutInMs);
587         if (error != E_ok)
588         {
589             errorResponseOut = "Error connecting to " + hostname + ":" + std::to_string(port);
590             sock.close();
591             goto done;
592         }
593 
594         // Initialize client.
595         error = initClient(timeoutInMs, errorResponseOut);
596         if (error == E_ok)
597         {
```

```
598             connectionStr = hostname + ":" + std::to_string(port);
599         }
600         else
601         {
602             disconnect();
603         }
604 
605         // Return error code (if any).
606     done:
607         if (verbose)
608             log.info("IrisClient::connect() leave (%s)\n", irisErrorCodeCStr(error));
609         return error;
610     }
611 
615     void connectSocketFd(SocketFd socketfd, unsigned timeoutInMs = 1000)
616     {
617         assert(mode == IRIS_TCP_CLIENT);
618 
619         if (verbose)
620             log.info("IrisClient::connectSocketFd(socketfd=%llu, timeout=%u)\n", (long long)socketfd,
    timeoutInMs);
621 
622         // Already connected?
623         std::string errorResponse;
624         IrisErrorCode error = E_ok;
625         if (adapter.isConnected() || sock.isConnected())
626         {
627             throw IrisErrorExceptionString(E_already_connected, "Already connected.");
628         }
629 
630         sock.setSocketFd(socketfd);
631         sock.setNonBlocking();
632 
633         // Unblock a potentially blocked worker thread which so far is waiting indefinitely
634         // on 'no socket'. This thread will block again on the socket we just created.
635         socketSet.stopWaitForEvent();
636 
637         // Initialize client.
638         error = initClient(timeoutInMs, errorResponse);
639         if (error != E_ok)
640         {
641             disconnect();
642             throw IrisErrorExceptionString(error, errorResponse);
643         }
644 
645         connectionStr = "(connected via socketfd)";
646     }
647 
651     IrisErrorCode disconnect()
652     {
653         if (verbose)
654         {
655             log.info("IrisClient::disconnect()\n");
656         }
657 
658         // Tell IrisInstance to stop sending requests to us.
659         // All Iris calls (including the inevitable final
660         // instanceRegistry_unregisterInstance()) will return
661         // E_not_connected from now on.
662         setConnectionInterface(nullptr);
663 
664         connectionStr = "(not connected)";
665 
666         if (mode != IRIS_TCP_CLIENT)
667         {
668             return E_ok;
669         }
670 
671         IrisErrorCode errorCode = E_ok;
672         {
673             // We just close the TCP connection. This is a first-class operation which always must be
    handled gracefully by the server.
674             // The server needs to do all cleanup automatically.
675             if (adapter.isConnected())
676                 errorCode = adapter.closeConnection();
677             if (sock.isConnected())
678             {
679                 if (errorCode != E_ok)
680                     sock.close();
681                 else
682                     errorCode = sock.close();
683             }
684         }
685 
686         // Wake up processing thread since there is no point to wait on a closed socket.
687         socketSet.stopWaitForEvent();
688 
```

```
689         return errorCode;
690     }
691
693     bool isConnected() const
694     {
695         return adapter.isConnected();
696     }
697
699     IrisInterface* getSendingInterface()
700     {
701         return this;
702     }
703
706     void setInstanceName(const std::string& instName)
707     {
708         if (isRegistered())
709         {
710             throw IrisErrorExceptionString(E_instance_already_registered, "IrisClient::setInstanceName()
    must be called before connect().");
711         }
712         irisInstanceInstName = instName;
713     }
714
717     IrisInstance& getIrisInstance() { return *this; }
718
721     void setSleepOnDestructionMs(uint64_t sleepOnDestructionMs_)
722     {
723         sleepOnDestructionMs = sleepOnDestructionMs_;
724     }
725
726
727     // --- IrisProcessEventsInterface implementation ---
728
745     virtual void processEvents() override
746     {
747         if (verbose >= 2)
748             log.info("IrisClient::processEvents() enter\n");
749
750         // in IRIS_SERVICE_SERVER mode, the adapter should work as server and hence call
751         // function processEventsServer()
752         switch (mode)
753         {
754         case IRIS_TCP_CLIENT:
755             adapter.processEventsClient();
756             break;
757         case IRIS_SERVICE_SERVER:
758             adapter.processEventsServer();
759             break;
760         }
761
762         if (verbose >= 2)
763             log.info("IrisClient::processEvents() leave\n");
764     }
765
769     virtual void waitForEvent() override
770     {
771         if (verbose >= 2)
772             log.info("IrisClient::waitForEvent() enter\n");
773         socketSet.waitForEvent(1000);
774         if (verbose >= 2)
775             log.info("IrisClient::waitForEvent() leave\n");
776     }
777
780     virtual void stopWaitForEvent() override
781     {
782         if (verbose)
783             log.info("IrisClient::stopWaitForEvent()\n");
784         socketSet.stopWaitForEvent();
785     }
786
788     void setPreferredSendingFormat(impl::IrisRpcAdapterTcp::Format p)
789     {
790         adapter.setPreferredSendingFormat(p);
791     }
792
794     impl::IrisRpcAdapterTcp::Format getEffectiveSendingFormat() const
795     {
796         return adapter.getEffectiveSendingFormat();
797     }
798
800     void setVerbose(unsigned level, bool increaseOnly = false)
801     {
802         if (increaseOnly && (level <= verbose))
803         {
804             return;
805         }
806
```

```
807         verbose = level;
808         if (verbose)
809             log.info("IrisClient: verbose logging enabled (level %d)\n", verbose);
810         if (mode == IRIS_TCP_CLIENT)
811         {
812             sock.setVerbose(verbose);
813         }
814         socketSet.setVerbose(verbose);
815         if (verbose)
816         {
817             log.setIrisMessageLogLevelFlags(IrisLogger::TIMESTAMP);
818         }
819     }
820
822     void setIrisMessageLogLevel(unsigned level, bool increaseOnly = false)
823     {
824         if (increaseOnly && (level <= irisMessageLogLevel))
825         {
826             return;
827         }
828
829         irisMessageLogLevel = level;
830         log.setIrisMessageLogLevel(irisMessageLogLevel);
831     }
832
834     std::string getConnectionStr() const { return connectionStr; }
835
836 private:
838     enum Mode
839     {
840         IRIS_TCP_CLIENT,
841         IRIS_SERVICE_SERVER
842     };
843
844     // Shared code for constructors in client mode.
845     void init(Mode mode_, const std::string& instName)
846     {
847         log.setLogContext("IrisTC");
848         mode = mode_;
849
850         // Set instance name of contained IrisInstance.
851         if (instName.empty())
852         {
853             setInstanceName("client.IrisClient");
854         }
855         else
856         {
857             setInstanceName(instName);
858         }
859
860         // Enable verbose logging?
861         setVerbose(static_cast<unsigned>(getEnvU64("IRIS_TCP_CLIENT_VERBOSE")), true);
862         irisMessageLogLevel = unsigned(getEnvU64("IRIS_TCP_CLIENT_LOG_MESSAGES"));
863         log.setIrisMessageLogLevel(irisMessageLogLevel);
864         log.setIrisMessageGetInstNameFunc([&](InstanceId instId){ return getInstName(instId); });
865
866         if (mode == IRIS_TCP_CLIENT)
867         {
868             socketSet.addSocket(&sock);
869         }
870         sendingInterface = adapter.getSendingInterface();
871
872         // Intercept all calls to the global instance since we must modify
    instanceRegistry_registerInstance() and
873         // instanceRegistry_unregisterInstance() and their responses.
874         instIdToInterface.push_back(&globalInstanceSendingInterface); // This must be index 0 in the
    vector (instId 0 == global instance).
875
876         if (mode == IRIS_SERVICE_SERVER)
877         {
878             socket_thread = std::unique_ptr<impl::IrisProcessEventsThread>(new
    impl::IrisProcessEventsThread(this, "TcpSocket"));
879         }
880     }
881
883     IrisErrorCode initClient(unsigned timeoutInMs, std::string& errorResponseOut)
884     {
885         assert(mode == IRIS_TCP_CLIENT);
886
887         // Initialize IrisRpcAdapterTcp.
888         try
889         {
890             adapter.initClient(&sock, &socketSet, &receivingInterface, verbose);
891         }
892         catch (const IrisErrorException& e)
893         {
894             if (e.getMessage().empty())
```

```
895             {
896                 throw IrisErrorExceptionString(e.getErrorCode(), "Client: Error connecting to server
    socket.");
897             }
898             else
899             {
900                 throw;
901             }
902         }
903 
904         // Handshake.
905         IrisErrorCode error = adapter.handshakeClient(errorResponseOut, timeoutInMs);
906 
907         if (error)
908         {
909             // We either tried to connect to an incompatible client or
910             // starting the model command line failed in spawnAndConnect().
911             // Simply returning the error code and the error message will throw an error exception in
    connectSocketFd().
912             // and main() will usually print it.
913             return error;
914         }
915 
916         // Start a thread to process incoming data in the background.
917         socket_thread = std::unique_ptr<impl::IrisProcessEventsThread>(new
    impl::IrisProcessEventsThread(this, "TcpSocket"));
918 
919         // Initialize IrisInstance.
920         setConnectionInterface(this);
921         registerInstance(irisInstanceInstName, iris::IrisInstance::UNIQUIFY |
    iris::IrisInstance::THROW_ON_ERROR);
922 
923         return error;
924     }
925 
927     void destructor()
928     {
929         if (childPid)
930         {
931             // Disconnect from spawned model and wait for the child process to exit.
932             disconnectAndWaitForChildToExit();
933         }
934         else
935         {
936             // Disconnect TCP connection.
937             disconnect();
938         }
939 
940         // Do not rely on destructor order. The socket_thread expects this
941         // object to be fully alive.
942         if (socket_thread)
943         {
944             socket_thread->terminate();
945         }
946 
947         switch (mode)
948         {
949         case IRIS_TCP_CLIENT:
950             socketSet.removeSocket(&sock);
951             break;
952 
953         case IRIS_SERVICE_SERVER:
954             socketSet.removeSocket(service_socket);
955             delete service_socket;
956             break;
957         }
958 
959         delete socketpairPoolToDelete;
960 
961         // Just to beautify stdout. No functional impact and usually 0.
962         iris::sleepMs(sleepOnDestructionMs);
963     }
964 
966     void connectCommandLineInternal(std::vector<std::string>& commandLine, const std::string&
    programName, bool keepOtherArgs)
967     {
968         // Parse client and server args.
969         IrisCommaSeparatedParameters clientArgs;
970         IrisCommaSeparatedParameters serverArgs;
971         std::vector<std::string> modelCommandLine;
972         for (size_t i = 0; i < commandLine.size(); i++)
973         {
974             if (commandLine[i] == "--")
975             {
976                 // Stop parsing args at "--".
977                 // The model command line follows.
978                 modelCommandLine.insert(modelCommandLine.begin(), commandLine.begin() + i + 1,
```

```
commandLine.end());
979                 commandLine.resize(i);
980                 clientArgs.set("spawn");
981                 break;
982             }
983 
984             // Get key of key[=value] args.
985             std::string key = commandLine[i];
986             size_t pos = key.find('=');
987             if (pos != std::string::npos)
988             {
989                 key = key.substr(0, pos);
990             }
991 
992             // Set client args.
993             if ((key == "spawn") || (key == "tcp") || (key == "port") || (key == "timeout") || (key ==
"iris-log") || (key == "verbose") || (key == "help"))
994             {
995                 clientArgs.set(commandLine[i]);
996                 commandLine.erase(commandLine.begin() + i--);
997             }
998             // Set server args.
999             else if (key == "server_verbose")
1000             {
1001                 serverArgs.set(commandLine[i].substr(7));
1002                 commandLine.erase(commandLine.begin() + i--);
1003             }
1004         }
1005 
1006         // Just print help? Overrides everything else.
1007         if (clientArgs.have("help"))
1008         {
1009             std::string help = getConnectCommandLineHelp();
1010             replaceString(help, "%prog", programName);
1011             throw IrisErrorExceptionString(E_help_message, help);
1012         }
1013 
1014         // Print errors for unknown args?
1015         if (!keepOtherArgs && !commandLine.empty())
1016         {
1017             throw IrisErrorExceptionString(E_error_message, "Unknown argument(s): " +
joinString(commandLine, ", ") + ".\n(Prepend \"--\" before model executable and its arguments.)");
1018         }
1019 
1020         if (clientArgs.have("spawn"))
1021         {
1022             clientArgs.erase("spawn");
1023 
1024             if (clientArgs.have("tcp"))
1025             {
1026                 throw IrisErrorExceptionString(E_error_message, "Only one out of \"--\" (start model)
and \"tcp\" may be specified.");
1027             }
1028 
1029             if (modelCommandLine.empty())
1030             {
1031                 throw IrisErrorExceptionString(E_error_message, "Missing/empty model command line.
Expected format: " + programName + " -- isim_system -C foo=bar. Try 'help'.");
1032             }
1033 
1034             // Start child process and connect to it using UNIX domain socket.
1035             spawnAndConnect(modelCommandLine, serverArgs.getParameterSpec(),
clientArgs.getParameterSpec());
1036         }
1037         else
1038         {
1039             // Connect via TCP. This is also the default if neither -- (start model) nor tcp are
specified. connect() needs an explicit "tcp" so set it here if not set.
1040             if (!clientArgs.have("tcp"))
1041             {
1042                 clientArgs.set("tcp");
1043             }
1044 
1045             if (clientArgs.have("port") && clientArgs.getStr("port").empty())
1046             {
1047                 throw IrisErrorExceptionString(E_error_message, "Missing argument for port. Expecting
port=N.");
1048             }
1049 
1050             if (!serverArgs.getMap().empty())
1051             {
1052                 throw IrisErrorExceptionString(E_error_message, "Server args cannot be set for
connections via \"tcp\". Specify server args on the model command line when starting the model.");
1053             }
1054 
1055             connect(clientArgs.getParameterSpec());
1056         }
```

```
1057     }
1058
1062     virtual void irisHandleMessage(const uint64_t* message) override
1063     {
1064         // Log message?
1065         if (irisMessageLogLevel)
1066         {
1067             log.irisMessage(message);
1068         }
1069
1070         // This calls one of these:
1071         // - this->globalInstanceSendingInterface_irisHandleMessage(); (for requests, instId == 0)
1072         // - Iris interface of a local instance (if a local instance talks to a local instance)
1073         // - sendingInterface (to send message to server using TCP)
1074         findInterface(IrisU64JsonReader::getInstId(message))->irisHandleMessage(message);
1075     }
1076
1079     void globalInstanceSendingInterface_irisHandleMessage(const uint64_t* message)
1080     {
1081         // This is only ever called for instId == 0.
1082         assert(IrisU64JsonReader::getInstId(message) == 0);
1083         assert(IrisU64JsonReader::isRequestOrNotification(message));
1084
1085         // Decode request.
1086         IrisU64JsonReader          r(message);
1087         IrisU64JsonReader::Request req    = r.openRequest();
1088         std::string                method = req.getMethod();
1089
1090         if (method == "instanceRegistry_registerInstance")
1091         {
1092             RequestId requestId = req.getRequestId();
1093
1094             // We received an instanceRegistry_registerInstance() request from a local instance:
1095             // - Create a new request id which is unique to this request for this TCP channel. (This is
     not required to be globally unique.)
1096             // - Allocate an ongoingInstanceRegistryCalls slot for this new request id and remember the
     original request id and params.channelId in it.
1097             // - Modify request id of request to the new request id so we can recognize the response
     later.
1098             // - Send modified request.
1099
1100             // Create a new request id which is unique to this request for this TCP channel. (This is
     not required to be globally unique.)
1101             RequestId newRequestId = generateNewRequestIdForRegisterInstanceCall();
1102
1103             // Get channelId.
1104             uint64_t channelId = IRIS_UINT64_MAX;
1105             if (!req.paramOptional(ISTR("channelId"), channelId))
1106             {
1107                 // Strange. 'params.channelId' is missing. This should never happen.
1108                 log.error(
1109                     "IrisClient::receivingInterface_irisHandleMessage():"
1110                     " Received instanceRegistry_registerInstance() request without channelId
     parameter:\n%s\n",
1111                     messageToString(message).c_str());
1112                 goto send;
1113             }
1114
1115             {
1116                 std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
1117                 // Allocate an ongoingInstanceRegistryCalls slot for this new request id and remember
     the
1118                 // original request id and params.channelId in it.
1119                 ongoingInstanceRegistryCalls[newRequestId] = OngoingInstanceRegistryCallEntry(method,
1120
     requestId,
1121
     channelId);
1122             }
1123
1124             // Create a modified reaquest that:
1125             //  - sets the new request id so we can recognize the response later.
1126             //  - removes the channelId parameter (it only has meaning in-process)
1127             IrisU64JsonReader original_message(message);
1128             IrisU64JsonWriter modified_message;
1129
1130             {
1131                 IrisU64JsonReader::Request original_req = original_message.openRequest();
1132
1133                 IrisU64JsonWriter::Request new_req =
     modified_message.openRequest(original_req.getMethod(),
1134
     original_req.getInstId());
1135                 new_req.setRequestId(newRequestId);
1136
1137                 std::string param;
1138                 while (original_req.readNextParam(param))
```

```
1139                {
1140                    if ((param == "channelId") || (param == "instId"))
1141                    {
1142                        // Skip the params we want to remove (channelId)
1143                        // and skip instId too because that will have already been filled in.
1144                        // skip over the value to the next parameter
1145                        original_message.skip();
1146                    }
1147                    else
1148                    {
1149                        new_req.paramSlow(param);
1150
1151                        // Pass through the original value
1152                        IrisValue value;
1153                        persist(original_message, value);
1154                        persist(modified_message, value);
1155                    }
1156                }
1157            }
1158
1159            // Send modified request.
1160            sendingInterface->irisHandleMessage(modified_message.getMessage());
1161            return;
1162        }
1163        else if (method == "instanceRegistry_unregisterInstance")
1164        {
1165            // We received an instanceRegistry_unregisterInstance() request from a local instance:
1166            // - Allocate an ongoingInstanceRegistryCalls slot for the request id and remember the
    instId of the unregistered instance in it.
1167            // - Send request unmodified.
1168
1169            // Get params.aInstId.
1170            InstanceId aInstId = IRIS_UINT64_MAX;
1171            if (!req.paramOptional(ISTR("aInstId"), aInstId))
1172            {
1173                // Strange. 'params.aInstId' is missing. This should never happen.
1174                log.error(
1175                    "IrisClient::receivingInterface_irisHandleMessage():"
1176                    " Received instanceRegistry_unregisterInstance() request without aInstId
    parameter:\n%s\n",
1177                    messageToString(message).c_str());
1178                goto send;
1179            }
1180
1181            if (!req.isNotification())
1182            {
1183                RequestId requestId = req.getRequestId();
1184
1185                if (aInstId == getCallerInstId(requestId))
1186                {
1187                    std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
1188                    // There will be a response to this request so we need to remember the interface to
    send it to.
1189                    // Allocate an ongoingInstanceRegistryCalls slot for the request id and remember
    the instId of the unregistered instance in it.
1190                    ongoingInstanceRegistryCalls[requestId] = OngoingInstanceRegistryCallEntry(method,
    aInstId);
1191                    goto send;
1192                }
1193            }
1194
1195            // There will be no more communication to the instance being unregistered.
1196            // Remove instance from instIdToInterface.
1197            assert(aInstId < InstanceId(instIdToInterface.size()));
1198            // sendingInterface: Forward messages to unknown instIds to the server. The global instance
    may have reassigned the same instId to some other instance behind the server which exists.
1199            instIdToInterface[aInstId] = sendingInterface;
1200
1201            // Intended fallthrough to send original request.
1202        }
1203        else if (method == "instanceRegistry_getList")
1204        {
1205            // We received an instanceRegistry_getList() request from a local instance:
1206            // - We want to remember/snoop all returned instance names we get in the response (for
    logging).
1207            // - Allocate an ongoingInstanceRegistryCalls slot for the request id in order to recognize
    the response.
1208            // - Send request unmodified.
1209
1210            if (!req.isNotification())
1211            {
1212                RequestId requestId = req.getRequestId();
1213                std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
1214                ongoingInstanceRegistryCalls[requestId] = OngoingInstanceRegistryCallEntry(method);
1215            }
1216
1217            // Intended fallthrough to send original request.
```

```
1218         }
1219
1220     send:
1221         // Send original message.
1222         sendingInterface->irisHandleMessage(message);
1223     }
1224
1228     void receivingInterface_irisHandleResponse(const uint64_t* message)
1229     {
1230         {
1231             std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
1232
1233             if (!ongoingInstanceRegistryCalls.empty())
1234             {
1235                 // Slow path is only used while a instanceRegistry_registerInstance() or
     instanceRegistry_unregisterInstance()
1236                 // call is ongoing. This is usually only the case at startup and shutdown.
1237
1238                 // We need to check whether this is the response to either
1239                 // instanceRegistry_registerInstance() or
1240                 // instanceRegistry_unregisterInstance() or
1241                 // any other response.
1242
1243                 // Decode response.
1244                 IrisU64JsonReader           r(message);
1245                 IrisU64JsonReader::Response resp = r.openResponse();
1246                 RequestId requestId = resp.getRequestId();
1247
1248                 // Check whether this is a response to one of our pending requests.
1249                 OngoingInstanceRegistryCallMap::iterator i =
     ongoingInstanceRegistryCalls.find(requestId);
1250                 if (i == ongoingInstanceRegistryCalls.end())
1251                 {
1252                     goto send; // None of the pending responses. Handle in the normal way.
1253                 }
1254
1255                 if (i->second.method == "instanceRegistry_registerInstance")
1256                 {
1257                     // This is a response to a previous instanceRegistry_registerInstance() call:
1258
1259                     IrisInterface* responseIfPtr = channel_registry.getChannel(i->second.channelId);
1260
1261                     if (resp.isError())
1262                     {
1263                         // The call failed, pass on the message.
1264                         responseIfPtr->irisHandleMessage(message);
1265                     }
1266                     else
1267                     {
1268                         // The call succeeded:
1269                         // - add new instId to our local instance registry
1270                         // - translate request id back to the original request id
1271                         // - send this modified response to the caller
1272                         // - erase this entry in ongoingInstanceRegistryCalls
1273
1274                         // Add instance to instIdToInterface.
1275                         InstanceId newInstId;
1276                         if (!resp.getResultReader().openObject().memberOptional(ISTR("instId"),
     newInstId))
1277                         {
1278                             // Strange. 'result.instId' is missing. This should never happen.
1279                             log.error(
1280                                 "IrisClient::receivingInterface_irisHandleResponse():"
1281                                 " Received instanceRegistry_registerInstance() response without
     result.instId:\n%s\n",
1282                                 messageToString(message).c_str());
1283                         }
1284                         else
1285                         {
1286                             // This is a valid response for instanceRegistry_registerInstance(): Enter
     newInstId into instIdToInterface.
1287                             findInterface(newInstId);
1288                             instIdToInterface[newInstId] = responseIfPtr;
1289                         }
1290
1291                         // Remember instance name.
1292                         std::string newInstName;
1293                         if (resp.getResultReader().openObject().memberOptional(ISTR("instName"),
     newInstName))
1294                         {
1295                             setInstName(newInstId, newInstName);
1296                         }
1297
1298                         // Translate the id back to the id of the original request and use the
     responseIfPtr to send the response.
1299                         IrisU64JsonWriter modifiedMessageWriter;
1300                         modifiedMessageWriter.copyMessageAndModifyId(message, i->second.id);
```

```
1301
1302                        // Log message?
1303                        if (irisMessageLogLevel)
1304                        {
1305                            log.irisMessage(modifiedMessageWriter.getMessage());
1306                        }
1307
1308                        responseIfPtr->irisHandleMessage(modifiedMessageWriter.getMessage());
1309                    }
1310
1311                    // Remove ongoingInstanceRegistryCalls entry now that we have seen the response.
1312                    ongoingInstanceRegistryCalls.erase(i);
1313                    return;
1314                }
1315                else if (i->second.method == "instanceRegistry_unregisterInstance")
1316                {
1317                    // This is a response to a previous instanceRegistry_unregisterInstance() call:
1318                    // - remove this instId from our local instance registry
1319                    // - remove this entry from ongoingInstanceRegistryCalls
1320                    // - send response to caller
1321
1322                    InstanceId aInstId = i->second.id;
1323
1324                    // Remeber the old response interface in case we need it after we override it
1325                    IrisInterface* aInst_responseIf = instIdToInterface[aInstId];
1326
1327                    // Remove instance from instIdToInterface.
1328                    assert(aInstId < InstanceId(instIdToInterface.size()));
1329                    // sendingInterface: Forward messages to unknown instIds to the server. The global
    instance may have reassigned the same instId to some other instance behind the server which exists.
1330                    instIdToInterface[aInstId] = sendingInterface;
1331                    setInstName(aInstId, ""); // IrisLogger will generate a default name for unknown
    instance ids.
1332                    // Remove ongoingInstanceRegistryCalls entry.
1333                    ongoingInstanceRegistryCalls.erase(i);
1334
1335                    if (aInstId == resp.getInstId())
1336                    {
1337                        // An instance unregistered itself so we need to call it directly rather than
1338                        // go through the normal message handler because we just set that to forward
1339                        // messages to this instId to the server.
1340                        aInst_responseIf->irisHandleMessage(message);
1341                        return;
1342                    }
1343
1344                    // Intended fallthrough to irisHandleMessage(message).
1345                }
1346                else if (i->second.method == "instanceRegistry_getList")
1347                {
1348                    // This is a response to a previous instanceRegistry_getList() call:
1349                    // - remember all instance names (for logging)
1350                    // - send response to caller
1351
1352                    // Remove ongoingInstanceRegistryCalls entry.
1353                    ongoingInstanceRegistryCalls.erase(i);
1354                    try
1355                    {
1356                        // Peek into instance list. We do not care whether this is just
1357                        // a subset of all instances or not. We take what we can get.
1358                        std::vector<InstanceInfo> instanceInfoList;
1359                        resp.getResult(instanceInfoList);
1360                        for (const auto& instanceInfo: instanceInfoList)
1361                        {
1362                            setInstName(instanceInfo.instId, instanceInfo.instName);
1363                        }
1364                    }
1365                    catch(const IrisErrorException&)
1366                    {
1367                        // Silently ignore bogus responses. The caller will handle the error.
1368                    }
1369                    // Intended fallthrough to irisHandleMessage(message).
1370                }
1371            }
1372        }
1373
1374    send:
1375        // Handle response in the normal way.
1376        irisHandleMessage(message);
1377    }
1378
1383    RequestId generateNewRequestIdForRegisterInstanceCall()
1384    {
1385        return nextInstIdForRegisterInstanceCall++;
1386    }
1387
1393    IrisInterface* findInterface(InstanceId instId)
1394    {
```

```
1395         if (instId >= IrisMaxTotalInstances)
1396         {
1397             log.error("IrisClient::findInterface(instId=0x%08x): got ridiculously high instId",
     int(instId));
1398             return sendingInterface;
1399         }
1400         if (instId >= InstanceId(instIdToInterface.size()))
1401         {
1402             instIdToInterface.resize(instId + 100, sendingInterface);
1403         }
1404         return instIdToInterface[instId];
1405     }
1406
1408     class GlobalInstanceSendingInterface : public IrisInterface
1409     {
1410     public:
1411         GlobalInstanceSendingInterface(IrisClient* parent_)
1412             : parent(parent_)
1413         {
1414         }
1415
1419         virtual void irisHandleMessage(const uint64_t* message) override
1420         {
1421             if (IrisU64JsonReader::isRequestOrNotification(message))
1422             {
1423                 // Intercept requests to the global instance so we can snoop on
1424                 // calls to instanceRegistry_registerInstance()
1425                 parent->globalInstanceSendingInterface_irisHandleMessage(message);
1426             }
1427             else
1428             {
1429                 // This is called for responses sent from clients to the global instance.
1430                 // Simply forward them as usual. Nothing to intercept.
1431                 parent->sendingInterface->irisHandleMessage(message);
1432             }
1433         }
1434
1435     private:
1436         IrisClient* const parent;
1437     };
1438
1440     class ReceivingInterface : public IrisInterface
1441     {
1442     public:
1443         ReceivingInterface(IrisLogger& log_, IrisClient* parent_)
1444             : parent(parent_)
1445             , log(log_)
1446         {
1447         }
1448
1450         virtual void irisHandleMessage(const uint64_t* message) override
1451         {
1452             InstanceId instId = IrisU64JsonReader::getInstId(message);
1453
1454             if (instId >= InstanceId(instId_to_thread_id.size()))
1455             {
1456                 // We do not have an entry for this instance therefore
1457                 // we have not been asked to marshal requests to a specific
1458                 // thread and should use the default.
1459                 // Pass thread id to IrisMessageQueue and IrisProcessEventsThread
1460                 setHandlerThread(instId, getDefaultThreadId());
1461             }
1462
1463             std::thread::id thread_id = instId_to_thread_id[instId];
1464             if (thread_id == std::this_thread::get_id())
1465             {
1466                 // Message has already been marshalled, forward on
1467                 if (IrisU64JsonReader::isRequestOrNotification(message))
1468                 {
1469                     parent->irisHandleMessage(message);
1470                 }
1471                 else
1472                 {
1473                     parent->receivingInterface_irisHandleResponse(message);
1474                 }
1475             }
1476             else
1477             {
1478                 message_queue.push(message, thread_id);
1479             }
1480         }
1481
1482         void setHandlerThread(InstanceId instId, std::thread::id thread_id)
1483         {
1484             if (instId >= IrisMaxTotalInstances)
1485             {
1486                 log.error(
```

```
1487                     "IrisClient::ReceivingInterface::setHandlerThread(instId=0x%08x):"
1488                     " got ridiculously high instId",
1489                     int(instId));
1490             }
1491             else if (instId >= InstanceId(instId_to_thread_id.size()))
1492             {
1493                 instId_to_thread_id.resize(instId + 100, getDefaultThreadId());
1494             }
1495
1496             instId_to_thread_id[instId] = thread_id;
1497         }
1498
1499         IrisErrorCode processMessagesForCurrentThread(bool waitForAMessage)
1500         {
1501             if (waitForAMessage)
1502             {
1503                 IrisErrorCode code = message_queue.waitForMessageForCurrentThread();
1504                 if (code != E_ok)
1505                 {
1506                     return code;
1507                 }
1508             }
1509             message_queue.processRequestsForCurrentThread();
1510
1511             return E_ok;
1512         }
1513
1514     private:
1516         std::thread::id getDefaultThreadId()
1517         {
1518             return process_events_thread.getThreadId();
1519         }
1520
1521         IrisClient* const parent;
1522
1524         impl::IrisMessageQueue message_queue{this};
1525
1527         std::vector<std::thread::id> instId_to_thread_id;
1528
1529         IrisLogger& log;
1530
1532         impl::IrisProcessEventsThread process_events_thread{&message_queue, "ClientMsgHandlr"};
1533     };
1534
1535 public: // IrisConnectionInterface
1536     virtual uint64_t registerIrisInterfaceChannel(IrisInterface* iris_interface, const std::string&
      connectionInfo) override
1537     {
1538         return channel_registry.registerChannel(iris_interface, connectionInfo);
1539     }
1540
1541     virtual void unregisterIrisInterfaceChannel(uint64_t channelId) override
1542     {
1543         IrisInterface* if_to_remove = channel_registry.getChannel(channelId);
1544
1545         std::vector<InstanceId> instIds_for_channel;
1546
1547         for (size_t i = 0; i < instIdToInterface.size(); i++)
1548         {
1549             if (instIdToInterface[i] == if_to_remove)
1550             {
1551                 InstanceId instId = InstanceId(i);
1552                 instIds_for_channel.push_back(instId);
1553             }
1554         }
1555         if (instIds_for_channel.size() > 0)
1556         {
1557             // Create an instance to call instanceRegistry_unregisterInstance() with.
1558             IrisInstance instance_killer(this, "framework.IrisClient.instance_killer",
1559                                          IrisInstance::UNIQUIFY);
1560             for (InstanceId instId : instIds_for_channel)
1561             {
1562                 instance_killer.irisCall().instanceRegistry_unregisterInstance(instId);
1563             }
1564         }
1565
1566         channel_registry.unregisterChannel(channelId);
1567     }
1568
1569     virtual IrisErrorCode processAsyncMessages(bool waitForAMessage) override
1570     {
1571         return receivingInterface.processMessagesForCurrentThread(waitForAMessage);
1572     }
1573
1574     virtual IrisInterface* getIrisInterface() override
1575     {
1576         return this;
```

```
1577     }
1578
1579     void unregisterChannel(uint64_t channelId)
1580     {
1581         channel_registry.unregisterChannel(channelId);
1582     }
1583
1584     // function called by class IrisPlugin
1585     uint64_t registerChannel(IrisC_CommunicationChannel* channel, const ::std::string& connectionInfo)
1586     {
1587         return channel_registry.registerChannel(channel, connectionInfo);
1588     }
1589
1590 public:
1593     void loadPlugin(const std::string& plugin_name)
1594     {
1595         assert(mode == IRIS_SERVICE_SERVER);
1596         assert(plugin == nullptr);
1597         plugin = std::unique_ptr<impl::IrisPlugin<IrisClient»(new impl::IrisPlugin<IrisClient>(this,
      plugin_name));
1598     }
1599
1600     void unloadPlugin()
1601     {
1602         assert(mode == IRIS_SERVICE_SERVER);
1603         plugin = nullptr;
1604     }
1605
1612     void initServiceServer(impl::IrisTcpSocket* socket_)
1613     {
1614         assert(mode == IRIS_SERVICE_SERVER);
1615         service_socket = socket_;
1616         socketSet.addSocket(service_socket);
1617         adapter.initServiceServer(service_socket, &socketSet, &receivingInterface, verbose);
1618     }
1619
1620 private:
1622     std::string getInstName(InstanceId instId)
1623     {
1624         // IrisLogger will generate a default name for unknown instances (empty string).
1625         return instId < instIdToInstName.size() ? instIdToInstName[instId] : std::string();
1626     }
1627
1629     void setInstName(InstanceId instId, const std::string& instName)
1630     {
1631         // Ignore ridiculously high instIds (programming errors).
1632         if (instId >= IrisMaxTotalInstances)
1633         {
1634             return;
1635         }
1636
1637         if (instId >= instIdToInstName.size())
1638         {
1639             instIdToInstName.resize(instId + 1, "");
1640         }
1641
1642         instIdToInstName[instId] = instName;
1643     }
1644
1645     // --- Private data. ---
1646
1648     IrisLogger log;
1649
1651     std::string irisInstanceInstName;
1652
1654     GlobalInstanceSendingInterface globalInstanceSendingInterface{this};
1655
1657     ReceivingInterface receivingInterface{log, this};
1658
1660     impl::IrisTcpSocket sock{log, 0};
1661
1663     impl::IrisTcpSocket* service_socket{nullptr};
1664
1666     impl::IrisTcpSocketSet socketSet{log, 0};
1667
1669     std::vector<IrisInterface*> instIdToInterface;
1670
1672     std::vector<std::string> instIdToInstName;
1673
1675     impl::IrisChannelRegistry channel_registry{log};
1676
1678     IrisInterface* sendingInterface{nullptr};
1679
1682     uint32_t nextInstIdForRegisterInstanceCall{0};
1683
1685     struct OngoingInstanceRegistryCallEntry
1686     {
```

```
1687         OngoingInstanceRegistryCallEntry()
1688         {
1689         }
1690 
1691         OngoingInstanceRegistryCallEntry(const std::string& method_, uint64_t id_ = IRIS_UINT64_MAX,
1692                                          uint64_t channelId_ = IRIS_UINT64_MAX)
1693             : method(method_)
1694             , id(id_)
1695             , channelId(channelId_)
1696         {
1697         }
1698 
1699         std::string method;                 // instanceRegistry_registerInstance,
      instanceRegistry_unregisterInstance or instanceRegistry_getList().
1700         uint64_t id{IRIS_UINT64_MAX};       // For instanceRegistry_registerInstance(): Original
      request id. For instanceRegistry_unregisterInstance(): params.aInstId.
1701         uint64_t channelId{IRIS_UINT64_MAX}; // For instanceRegistry_registerInstance() only:
      params.channelId.
1702     };
1703 
1707     typedef std::map<uint64_t, OngoingInstanceRegistryCallEntry> OngoingInstanceRegistryCallMap;
1708 
1709     OngoingInstanceRegistryCallMap ongoingInstanceRegistryCalls;
1710 
1712     std::mutex ongoingInstanceRegistryCallsMutex;
1713 
1715     unsigned verbose{0};
1716 
1718     unsigned irisMessageLogLevel{0};
1719 
1721     impl::IrisRpcAdapterTcp adapter{log};
1722 
1724     std::unique_ptr<impl::IrisProcessEventsThread> socket_thread{nullptr};
1725 
1727     Mode mode;
1728 
1730     std::string component_name;
1731 
1733     std::unique_ptr<impl::IrisPlugin<IrisClient» plugin{nullptr};
1734 
1736     std::string connectionStr{"(not connected)"};
1737 
1740     uint64_t sleepOnDestructionMs{};
1741 
1743     uint64_t childPid{};
1744 
1746     int lastExitStatus{-1};
1747 
1748     SocketpairPool *socketpairPool{};
1749 
1750     SocketpairPool *socketpairPoolToDelete{};
1751 };
1752 
1753 NAMESPACE_IRIS_END
1754 
1755 #endif // #ifndef ARM_INCLUDE_IrisClient_h
```

9.7 IrisCommandLineParser.h File Reference

Generic command line parser.

```
#include <cstdint>
#include <map>
#include <string>
#include <vector>
#include <functional>
#include <exception>
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisErrorException.h"
```

Classes

- class iris::IrisCommandLineParser
- struct iris::IrisCommandLineParser::Option

  Option container.

9.7.1 Detailed Description

Generic command line parser.

Copyright

9.8 IrisCommandLineParser.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisCommandLineParser_h
8 #define ARM_INCLUDE_IrisCommandLineParser_h
9
10 #include <cstdint>
11 #include <map>
12 #include <string>
13 #include <vector>
14 #include <functional>
15 #include <exception>
16
17 #include "iris/detail/IrisCommon.h"
18 #include "iris/detail/IrisErrorException.h"
19
20 NAMESPACE_IRIS_START
21
33 #if 0
34 #include <iostream>
35 #include "iris/IrisCommandLineParser.h"
36
37 int main(int argc, const char* argv[])
38 {
39     // Declare command line options.
40     iris::IrisCommandLineParser options("mytool", "Usage: mytool [OPTIONS]\n", "0.0.1");
41     options.addOption('v', "verbose", "Be more verbose (may be specified multiple times)."); // Switch
      option.
42     options.addOption(0, "port", "Specify local server port.", "PORT", "7999"); // Option with argument,
      without a short option.
43
44     // Parse command line.
45     options.parseCommandLine(argc, argv);
46
47     // Use options.
48     if (options.getSwitch("verbose"))
49     {
50         std::cout « "Verbose level: " « options.getSwitch("verbose") « "\n";
51     }
52     std::cout « "Port:  " « options.getInt("port") « "\n";
53     return 0;
54 }
55 #endif
56 class IrisCommandLineParser
57 {
58 public:
60     static const bool KeepDashDash = true;
61
63     struct Option
64     {
65         // Public interface:
66
71         Option& setList(char sep = ',') { listSeparator = sep; return *this; }
72
73     private:
74         // Meta info:
75
77         char shortOption{};
78
81         std::string longOption;
82
84         std::string help;
85
89         std::string formalArgumentName;
90
92         std::string defaultValue;
93
95         char listSeparator{};
96
98         bool hasFormalArgument() const { return !formalArgumentName.empty(); }
99
100         // Actual values from command line:
101
105         std::string value;
```

```
106
108         bool isSpecified{};
109
111         void setValue(const std::string& v);
112
114         void unsetValue();
115
116         friend class IrisCommandLineParser;
117     };
118
122     IrisCommandLineParser(const std::string& programName, const std::string& usageHeader, const
    std::string& versionStr, bool keepDashDash = false);
123
131     Option& addOption(char shortOption, const std::string& longOption, const std::string& help, const
    std::string& formalArgumentName = std::string(), const std::string& defaultValue = std::string());
132
135     Option& addOption(char shortOption, const std::string& longOption, const std::string& help, const
    std::string& formalArgumentName, int64_t defaultValue)
136     {
137         return addOption(shortOption, longOption, help, formalArgumentName,
    std::to_string(defaultValue));
138     }
139
162     int parseCommandLine(int argc, const char** argv);
163     int parseCommandLine(int argc, char** argv) { return parseCommandLine(argc, const_cast<const
    char**>(argv)); }
164
167     void noNonOptionArguments();
168
172     void pleaseSpecifyOneOf(const std::vector<std::string>& options, const std::vector<std::string>&
    formalNonOptionArguments = std::vector<std::string>());
173
175     std::string getStr(const std::string& longOption) const;
176
179     int64_t getInt(const std::string& longOption) const;
180
183     uint64_t getUint(const std::string& longOption) const;
184
187     double getDbl(const std::string& longOption) const;
188
190     uint64_t getSwitch(const std::string& longOption) const;
191
193     bool operator()(const std::string& longOption) const { return getSwitch(longOption) > 0; }
194
196     std::vector<std::string> getList(const std::string& longOption) const;
197
201     std::map<std::string,std::string> getMap(const std::string& longOption) const;
202
206     bool isSpecified(const std::string& longOption) const;
207
209     const std::vector<std::string>& getNonOptionArguments() const { return nonOptionArguments; }
210
213     std::vector<std::string>& getNonOptionArguments() { return nonOptionArguments; }
214
218     void clear();
219
224     int printMessage(const std::string& message, int error = 0, bool exit = false) const;
225
227     int printError(const std::string& message) const;
228
232     int printErrorAndExit(const std::string& message) const;
233
237     int printErrorAndExit(const std::exception& e) const;
238
241     void throwError(const std::string& message) const
242     {
243         throw IrisErrorExceptionString(E_error_message, message);
244     }
245
257     void setMessageFunc(const std::function<int(const std::string& message, int error, bool exit)>&
    messageFunc);
258
262     static int defaultMessageFunc(const std::string& message, int error, bool exit);
263
267     std::string getHelpMessage() const;
268
272     void setValue(const std::string& longOption, const std::string& value, bool append = false);
273
276     void unsetValue(const std::string& longOption);
277
279     void setProgramName(const std::string& programName_, bool append = false);
280
282     std::string getProgramName() const { return programName; }
283
285     void setHelpMessagePad(uint64_t pad) { helpMessagePad = pad;}
286
287 private:
```

```
290     Option& getOption(const std::string& longOption);
291
293     const Option& getOption(const std::string& longOption) const;
294
296     std::string programName;
297
299     std::string usageHeader;
300
302     std::string versionStr;
303
305     bool keepDashDash;
306
308     std::vector<std::string> optionList;
309
312     std::map<std::string, Option> options;
313
315     std::vector<std::string> nonOptionArguments;
316
318     size_t helpMessagePad{20};
319
321     std::function<int(const std::string& message, int error, bool exit)> messageFunc;
322 };
323
324 NAMESPACE_IRIS_END
325
326 #endif // ARM_INCLUDE_IrisCommandLineParser_h
```

9.9 IrisElfDwarfArm.h File Reference

Constants for the register.canonicalRnScheme "ElfDwarf" for architecture Arm.

```
#include "iris/detail/IrisInterface.h"
#include "iris/detail/IrisCommon.h"
```

Enumerations

- enum **ElfDwarfArm** : uint64_t {
  ARM_R0 = 0x2800000000 , **ARM_R1** = 0x2800000001 , **ARM_R2** = 0x2800000002 , **ARM_R3** = 0x2800000003 ,
  ARM_R4 = 0x2800000004 , **ARM_R5** = 0x2800000005 , **ARM_R6** = 0x2800000006 , **ARM_R7** = 0x2800000007 ,
  ARM_R8 = 0x2800000008 , **ARM_R9** = 0x2800000009 , **ARM_R10** = 0x280000000a , **ARM_R11** = 0x280000000b ,
  ARM_R12 = 0x280000000c , **ARM_R13** = 0x280000000d , **ARM_R14** = 0x280000000e , **ARM_R15** = 0x280000000f ,
  ARM_SPSR = 0x2800000080 , **ARM_SPSR_fiq** = 0x2800000081 , **ARM_SPSR_irq** = 0x2800000082 , **ARM_SPSR_abt** = 0x2800000083 ,
  ARM_SPSR_und = 0x2800000084 , **ARM_SPSR_svc** = 0x2800000085 , **ARM_R8_fiq** = 0x2800000097 , **ARM_R9_fiq** = 0x2800000098 ,
  ARM_R10_fiq = 0x2800000099 , **ARM_R11_fiq** = 0x280000009a , **ARM_R12_fiq** = 0x280000009b , **ARM_R13_fiq** = 0x280000009c ,
  ARM_R14_fiq = 0x280000009d , **ARM_R13_irq** = 0x280000009e , **ARM_R14_irq** = 0x280000009f , **ARM_R13_abt** = 0x28000000a0 ,
  ARM_R14_abt = 0x28000000a1 , **ARM_R13_und** = 0x28000000a2 , **ARM_R14_und** = 0x28000000a3 , **ARM_R13_svc** = 0x28000000a4 ,
  ARM_R14_svc = 0x28000000a5 , **ARM_D0** = 0x2800000100 , **ARM_D1** = 0x2800000101 , **ARM_D2** = 0x2800000102 ,
  ARM_D3 = 0x2800000103 , **ARM_D4** = 0x2800000104 , **ARM_D5** = 0x2800000105 , **ARM_D6** = 0x2800000106 ,
  ARM_D7 = 0x2800000107 , **ARM_D8** = 0x2800000108 , **ARM_D9** = 0x2800000109 , **ARM_D10** = 0x280000010a ,
  ARM_D11 = 0x280000010b , **ARM_D12** = 0x280000010c , **ARM_D13** = 0x280000010d , **ARM_D14** = 0x280000010e ,
  ARM_D15 = 0x280000010f , **ARM_D16** = 0x2800000110 , **ARM_D17** = 0x2800000111 , **ARM_D18** = 0x2800000112 ,
  ARM_D19 = 0x2800000113 , **ARM_D20** = 0x2800000114 , **ARM_D21** = 0x2800000115 , **ARM_D22** =

0x2800000116 ,
ARM_D23 = 0x2800000117 , **ARM_D24** = 0x2800000118 , **ARM_D25** = 0x2800000119 , **ARM_D26** = 0x280000011a ,
ARM_D27 = 0x280000011b , **ARM_D28** = 0x280000011c , **ARM_D29** = 0x280000011d , **ARM_D30** = 0x280000011e ,
ARM_D31 = 0x280000011f , **AARCH64_X0** = 0xb700000000 , **AARCH64_X1** = 0xb700000001 , **AARCH64_X2** = 0xb700000002 ,
AARCH64_X3 = 0xb700000003 , **AARCH64_X4** = 0xb700000004 , **AARCH64_X5** = 0xb700000005 , **AARCH64_X6** = 0xb700000006 ,
AARCH64_X7 = 0xb700000007 , **AARCH64_X8** = 0xb700000008 , **AARCH64_X9** = 0xb700000009 , **AARCH64_X10** = 0xb70000000a ,
AARCH64_X11 = 0xb70000000b , **AARCH64_X12** = 0xb70000000c , **AARCH64_X13** = 0xb70000000d , **AARCH64_X14** = 0xb70000000e ,
AARCH64_X15 = 0xb70000000f , **AARCH64_X16** = 0xb700000010 , **AARCH64_X17** = 0xb700000011 , **AARCH64_X18** = 0xb700000012 ,
AARCH64_X19 = 0xb700000013 , **AARCH64_X20** = 0xb700000014 , **AARCH64_X21** = 0xb700000015 , **AARCH64_X22** = 0xb700000016 ,
AARCH64_X23 = 0xb700000017 , **AARCH64_X24** = 0xb700000018 , **AARCH64_X25** = 0xb700000019 , **AARCH64_X26** = 0xb70000001a ,
AARCH64_X27 = 0xb70000001b , **AARCH64_X28** = 0xb70000001c , **AARCH64_X29** = 0xb70000001d , **AARCH64_X30** = 0xb70000001e ,
AARCH64_SP = 0xb70000001f , **AARCH64_ELR** = 0xb700000021 , **AARCH64_V0** = 0xb700000040 , **AARCH64_V1** = 0xb700000041 ,
AARCH64_V2 = 0xb700000042 , **AARCH64_V3** = 0xb700000043 , **AARCH64_V4** = 0xb700000044 , **AARCH64_V5** = 0xb700000045 ,
AARCH64_V6 = 0xb700000046 , **AARCH64_V7** = 0xb700000047 , **AARCH64_V8** = 0xb700000048 , **AARCH64_V9** = 0xb700000049 ,
AARCH64_V10 = 0xb70000004a , **AARCH64_V11** = 0xb70000004b , **AARCH64_V12** = 0xb70000004c , **AARCH64_V13** = 0xb70000004d ,
AARCH64_V14 = 0xb70000004e , **AARCH64_V15** = 0xb70000004f , **AARCH64_V16** = 0xb700000050 , **AARCH64_V17** = 0xb700000051 ,
AARCH64_V18 = 0xb700000052 , **AARCH64_V19** = 0xb700000053 , **AARCH64_V20** = 0xb700000054 , **AARCH64_V21** = 0xb700000055 ,
AARCH64_V22 = 0xb700000056 , **AARCH64_V23** = 0xb700000057 , **AARCH64_V24** = 0xb700000058 , **AARCH64_V25** = 0xb700000059 ,
AARCH64_V26 = 0xb70000005a , **AARCH64_V27** = 0xb70000005b , **AARCH64_V28** = 0xb70000005c , **AARCH64_V29** = 0xb70000005d ,
AARCH64_V30 = 0xb70000005e , **AARCH64_V31** = 0xb70000005f }

9.9.1 Detailed Description

Constants for the register.canonicalRnScheme "ElfDwarf" for architecture Arm.

Date

Copyright ARM Limited 2019. All Rights Reserved.

9.10 IrisElfDwarfArm.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisElfDwarfArm_h
9 #define ARM_INCLUDE_IrisElfDwarfArm_h
10
11 #include "iris/detail/IrisInterface.h" // uint64_t
12 #include "iris/detail/IrisCommon.h"    // namespace iris
13
14 NAMESPACE_IRIS_START
15
16 namespace ElfDwarf
17 {
18
19 enum ElfDwarfArm: uint64_t
```

```
20 {
21 //  Constant       canonicalRn      Register     Architecture ELF-Arch DwarfReg
22 //  ================================================================================
23     ARM_R0       = 0x2800000000, // R0           EM_ARM              40         0
24     ARM_R1       = 0x2800000001, // R1           EM_ARM              40         1
25     ARM_R2       = 0x2800000002, // R2           EM_ARM              40         2
26     ARM_R3       = 0x2800000003, // R3           EM_ARM              40         3
27     ARM_R4       = 0x2800000004, // R4           EM_ARM              40         4
28     ARM_R5       = 0x2800000005, // R5           EM_ARM              40         5
29     ARM_R6       = 0x2800000006, // R6           EM_ARM              40         6
30     ARM_R7       = 0x2800000007, // R7           EM_ARM              40         7
31     ARM_R8       = 0x2800000008, // R8           EM_ARM              40         8
32     ARM_R9       = 0x2800000009, // R9           EM_ARM              40         9
33     ARM_R10      = 0x280000000a, // R10          EM_ARM              40        10
34     ARM_R11      = 0x280000000b, // R11          EM_ARM              40        11
35     ARM_R12      = 0x280000000c, // R12          EM_ARM              40        12
36     ARM_R13      = 0x280000000d, // R13          EM_ARM              40        13
37     ARM_R14      = 0x280000000e, // R14          EM_ARM              40        14
38     ARM_R15      = 0x280000000f, // R15          EM_ARM              40        15
39     ARM_SPSR     = 0x2800000080, // SPSR         EM_ARM              40       128
40     ARM_SPSR_fiq = 0x2800000081, // SPSR_fiq     EM_ARM              40       129
41     ARM_SPSR_irq = 0x2800000082, // SPSR_irq     EM_ARM              40       130
42     ARM_SPSR_abt = 0x2800000083, // SPSR_abt     EM_ARM              40       131
43     ARM_SPSR_und = 0x2800000084, // SPSR_und     EM_ARM              40       132
44     ARM_SPSR_svc = 0x2800000085, // SPSR_svc     EM_ARM              40       133
45     ARM_R8_fiq   = 0x2800000097, // R8_fiq       EM_ARM              40       151
46     ARM_R9_fiq   = 0x2800000098, // R9_fiq       EM_ARM              40       152
47     ARM_R10_fiq  = 0x2800000099, // R10_fiq      EM_ARM              40       153
48     ARM_R11_fiq  = 0x280000009a, // R11_fiq      EM_ARM              40       154
49     ARM_R12_fiq  = 0x280000009b, // R12_fiq      EM_ARM              40       155
50     ARM_R13_fiq  = 0x280000009c, // R13_fiq      EM_ARM              40       156
51     ARM_R14_fiq  = 0x280000009d, // R14_fiq      EM_ARM              40       157
52     ARM_R13_irq  = 0x280000009e, // R13_irq      EM_ARM              40       158
53     ARM_R14_irq  = 0x280000009f, // R14_irq      EM_ARM              40       159
54     ARM_R13_abt  = 0x28000000a0, // R13_abt      EM_ARM              40       160
55     ARM_R14_abt  = 0x28000000a1, // R14_abt      EM_ARM              40       161
56     ARM_R13_und  = 0x28000000a2, // R13_und      EM_ARM              40       162
57     ARM_R14_und  = 0x28000000a3, // R14_und      EM_ARM              40       163
58     ARM_R13_svc  = 0x28000000a4, // R13_svc      EM_ARM              40       164
59     ARM_R14_svc  = 0x28000000a5, // R14_svc      EM_ARM              40       165
60     ARM_D0       = 0x2800000100, // D0           EM_ARM              40       256
61     ARM_D1       = 0x2800000101, // D1           EM_ARM              40       257
62     ARM_D2       = 0x2800000102, // D2           EM_ARM              40       258
63     ARM_D3       = 0x2800000103, // D3           EM_ARM              40       259
64     ARM_D4       = 0x2800000104, // D4           EM_ARM              40       260
65     ARM_D5       = 0x2800000105, // D5           EM_ARM              40       261
66     ARM_D6       = 0x2800000106, // D6           EM_ARM              40       262
67     ARM_D7       = 0x2800000107, // D7           EM_ARM              40       263
68     ARM_D8       = 0x2800000108, // D8           EM_ARM              40       264
69     ARM_D9       = 0x2800000109, // D9           EM_ARM              40       265
70     ARM_D10      = 0x280000010a, // D10          EM_ARM              40       266
71     ARM_D11      = 0x280000010b, // D11          EM_ARM              40       267
72     ARM_D12      = 0x280000010c, // D12          EM_ARM              40       268
73     ARM_D13      = 0x280000010d, // D13          EM_ARM              40       269
74     ARM_D14      = 0x280000010e, // D14          EM_ARM              40       270
75     ARM_D15      = 0x280000010f, // D15          EM_ARM              40       271
76     ARM_D16      = 0x2800000110, // D16          EM_ARM              40       272
77     ARM_D17      = 0x2800000111, // D17          EM_ARM              40       273
78     ARM_D18      = 0x2800000112, // D18          EM_ARM              40       274
79     ARM_D19      = 0x2800000113, // D19          EM_ARM              40       275
80     ARM_D20      = 0x2800000114, // D20          EM_ARM              40       276
81     ARM_D21      = 0x2800000115, // D21          EM_ARM              40       277
82     ARM_D22      = 0x2800000116, // D22          EM_ARM              40       278
83     ARM_D23      = 0x2800000117, // D23          EM_ARM              40       279
84     ARM_D24      = 0x2800000118, // D24          EM_ARM              40       280
85     ARM_D25      = 0x2800000119, // D25          EM_ARM              40       281
86     ARM_D26      = 0x280000011a, // D26          EM_ARM              40       282
87     ARM_D27      = 0x280000011b, // D27          EM_ARM              40       283
88     ARM_D28      = 0x280000011c, // D28          EM_ARM              40       284
89     ARM_D29      = 0x280000011d, // D29          EM_ARM              40       285
90     ARM_D30      = 0x280000011e, // D30          EM_ARM              40       286
91     ARM_D31      = 0x280000011f, // D31          EM_ARM              40       287
92     AARCH64_X0   = 0xb700000000, // X0           EM_AARCH64         183         0
93     AARCH64_X1   = 0xb700000001, // X1           EM_AARCH64         183         1
94     AARCH64_X2   = 0xb700000002, // X2           EM_AARCH64         183         2
95     AARCH64_X3   = 0xb700000003, // X3           EM_AARCH64         183         3
96     AARCH64_X4   = 0xb700000004, // X4           EM_AARCH64         183         4
97     AARCH64_X5   = 0xb700000005, // X5           EM_AARCH64         183         5
98     AARCH64_X6   = 0xb700000006, // X6           EM_AARCH64         183         6
99     AARCH64_X7   = 0xb700000007, // X7           EM_AARCH64         183         7
100     AARCH64_X8   = 0xb700000008, // X8           EM_AARCH64         183         8
101     AARCH64_X9   = 0xb700000009, // X9           EM_AARCH64         183         9
102     AARCH64_X10  = 0xb70000000a, // X10          EM_AARCH64         183        10
103     AARCH64_X11  = 0xb70000000b, // X11          EM_AARCH64         183        11
104     AARCH64_X12  = 0xb70000000c, // X12          EM_AARCH64         183        12
105     AARCH64_X13  = 0xb70000000d, // X13          EM_AARCH64         183        13
106     AARCH64_X14  = 0xb70000000e, // X14          EM_AARCH64         183        14
```

```
107     AARCH64_X15  = 0xb70000000f, // X15          EM_AARCH64        183       15
108     AARCH64_X16  = 0xb700000010, // X16          EM_AARCH64        183       16
109     AARCH64_X17  = 0xb700000011, // X17          EM_AARCH64        183       17
110     AARCH64_X18  = 0xb700000012, // X18          EM_AARCH64        183       18
111     AARCH64_X19  = 0xb700000013, // X19          EM_AARCH64        183       19
112     AARCH64_X20  = 0xb700000014, // X20          EM_AARCH64        183       20
113     AARCH64_X21  = 0xb700000015, // X21          EM_AARCH64        183       21
114     AARCH64_X22  = 0xb700000016, // X22          EM_AARCH64        183       22
115     AARCH64_X23  = 0xb700000017, // X23          EM_AARCH64        183       23
116     AARCH64_X24  = 0xb700000018, // X24          EM_AARCH64        183       24
117     AARCH64_X25  = 0xb700000019, // X25          EM_AARCH64        183       25
118     AARCH64_X26  = 0xb70000001a, // X26          EM_AARCH64        183       26
119     AARCH64_X27  = 0xb70000001b, // X27          EM_AARCH64        183       27
120     AARCH64_X28  = 0xb70000001c, // X28          EM_AARCH64        183       28
121     AARCH64_X29  = 0xb70000001d, // X29          EM_AARCH64        183       29
122     AARCH64_X30  = 0xb70000001e, // X30          EM_AARCH64        183       30
123     AARCH64_SP   = 0xb70000001f, // SP           EM_AARCH64        183       31
124     AARCH64_ELR  = 0xb700000021, // ELR          EM_AARCH64        183       33
125     AARCH64_V0   = 0xb700000040, // V0           EM_AARCH64        183       64
126     AARCH64_V1   = 0xb700000041, // V1           EM_AARCH64        183       65
127     AARCH64_V2   = 0xb700000042, // V2           EM_AARCH64        183       66
128     AARCH64_V3   = 0xb700000043, // V3           EM_AARCH64        183       67
129     AARCH64_V4   = 0xb700000044, // V4           EM_AARCH64        183       68
130     AARCH64_V5   = 0xb700000045, // V5           EM_AARCH64        183       69
131     AARCH64_V6   = 0xb700000046, // V6           EM_AARCH64        183       70
132     AARCH64_V7   = 0xb700000047, // V7           EM_AARCH64        183       71
133     AARCH64_V8   = 0xb700000048, // V8           EM_AARCH64        183       72
134     AARCH64_V9   = 0xb700000049, // V9           EM_AARCH64        183       73
135     AARCH64_V10  = 0xb70000004a, // V10          EM_AARCH64        183       74
136     AARCH64_V11  = 0xb70000004b, // V11          EM_AARCH64        183       75
137     AARCH64_V12  = 0xb70000004c, // V12          EM_AARCH64        183       76
138     AARCH64_V13  = 0xb70000004d, // V13          EM_AARCH64        183       77
139     AARCH64_V14  = 0xb70000004e, // V14          EM_AARCH64        183       78
140     AARCH64_V15  = 0xb70000004f, // V15          EM_AARCH64        183       79
141     AARCH64_V16  = 0xb700000050, // V16          EM_AARCH64        183       80
142     AARCH64_V17  = 0xb700000051, // V17          EM_AARCH64        183       81
143     AARCH64_V18  = 0xb700000052, // V18          EM_AARCH64        183       82
144     AARCH64_V19  = 0xb700000053, // V19          EM_AARCH64        183       83
145     AARCH64_V20  = 0xb700000054, // V20          EM_AARCH64        183       84
146     AARCH64_V21  = 0xb700000055, // V21          EM_AARCH64        183       85
147     AARCH64_V22  = 0xb700000056, // V22          EM_AARCH64        183       86
148     AARCH64_V23  = 0xb700000057, // V23          EM_AARCH64        183       87
149     AARCH64_V24  = 0xb700000058, // V24          EM_AARCH64        183       88
150     AARCH64_V25  = 0xb700000059, // V25          EM_AARCH64        183       89
151     AARCH64_V26  = 0xb70000005a, // V26          EM_AARCH64        183       90
152     AARCH64_V27  = 0xb70000005b, // V27          EM_AARCH64        183       91
153     AARCH64_V28  = 0xb70000005c, // V28          EM_AARCH64        183       92
154     AARCH64_V29  = 0xb70000005d, // V29          EM_AARCH64        183       93
155     AARCH64_V30  = 0xb70000005e, // V30          EM_AARCH64        183       94
156     AARCH64_V31  = 0xb70000005f, // V31          EM_AARCH64        183       95
157 }; // enum ElfDwarfArm
158
159 } // namespace ElfDwarf
160
161 NAMESPACE_IRIS_END
162
163 #endif // ARM_INCLUDE_IrisElfDwarfArm_h
164
```

9.11 IrisEventEmitter.h File Reference

A utility class for emitting Iris events.

`#include "iris/detail/IrisEventEmitterBase.h"`

Classes

- class iris::IrisEventEmitter< ARGS >

  A helper class for generating Iris events.

9.11.1 Detailed Description

A utility class for emitting Iris events.

9.12 IrisEventEmitter.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisEventEmitter_h
9 #define ARM_INCLUDE_IrisEventEmitter_h
10
11 #include "iris/detail/IrisEventEmitterBase.h"
12
13 NAMESPACE_IRIS_START
14
35 template <typename... ARGS>
36 class IrisEventEmitter : public IrisEventEmitterBase
37 {
38 public:
40     IrisEventEmitter()
41         : IrisEventEmitterBase(sizeof...(ARGS))
42     {
43     }
44
52     void operator()(ARGS... args)
53     {
54         emitEvent(args...);
55     }
56 };
57
58 NAMESPACE_IRIS_END
59
60 #endif // ARM_INCLUDE_IrisEventEmitter_h
```

9.13 IrisGlobalInstance.h File Reference

Central instance which lives in the simulation engine and distributes all Iris messages.

```
#include "iris/IrisInstance.h"
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisFunctionDecoder.h"
#include "iris/detail/IrisInterface.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include "iris/detail/IrisReceivedRequest.h"
#include "iris/impl/IrisChannelRegistry.h"
#include "iris/impl/IrisPlugin.h"
#include "iris/impl/IrisServiceClient.h"
#include "iris/impl/IrisTcpServer.h"
#include <atomic>
#include <list>
#include <map>
#include <memory>
#include <mutex>
#include <string>
#include <thread>
#include <unordered_map>
#include <vector>
```

Classes

- class iris::IrisGlobalInstance

9.13.1 Detailed Description

Central instance which lives in the simulation engine and distributes all Iris messages.

Date

The IrisGlobalInstance lives in the simulation engine. It contains all central data structures like the instance registry. It is responsible for distributing Iris messages to all in-process instances and to the IrisTcpServer.

9.14 IrisGlobalInstance.h

Go to the documentation of this file.

```
1
10 #ifndef ARM_INCLUDE_IrisGlobalInstance_h
11 #define ARM_INCLUDE_IrisGlobalInstance_h
12
13 #include "iris/IrisInstance.h"
14 #include "iris/detail/IrisCommon.h"
15 #include "iris/detail/IrisFunctionDecoder.h"
16 #include "iris/detail/IrisInterface.h"
17 #include "iris/detail/IrisLogger.h"
18 #include "iris/detail/IrisObjects.h"
19 #include "iris/detail/IrisReceivedRequest.h"
20
21 #include "iris/impl/IrisChannelRegistry.h"
22 #include "iris/impl/IrisPlugin.h"
23 #include "iris/impl/IrisServiceClient.h"
24 #include "iris/impl/IrisTcpServer.h"
25
26 #include <atomic>
27 #include <list>
28 #include <map>
29 #include <memory>
30 #include <mutex>
31 #include <string>
32 #include <thread>
33 #include <unordered_map>
34 #include <vector>
35
36 NAMESPACE_IRIS_START
37
38 class IrisGlobalInstance : public IrisInterface
39     , public IrisConnectionInterface
40 {
41 public:
43     IrisGlobalInstance();
44
46     ~IrisGlobalInstance();
47
50     uint64_t registerChannel(IrisC_CommunicationChannel* channel, const std::string& connectionInfo);
51
53     void unregisterChannel(uint64_t channelId);
54
57     IrisInstance& getIrisInstance() { return irisInstance; }
58
59 public: // IrisConnectionInterface
64     virtual uint64_t registerIrisInterfaceChannel(IrisInterface* iris_interface, const std::string&
      connectionInfo) override;
65
68     virtual void unregisterIrisInterfaceChannel(uint64_t channelId) override
69     {
70         unregisterChannel(channelId);
71     }
72
73     virtual IrisErrorCode processAsyncMessages(bool waitForAMessage) override
74     {
75         return irisProxyInterface.load()->processAsyncMessagesInProxy(waitForAMessage);
76     }
77
79     virtual IrisInterface* getIrisInterface() override
80     {
81         return this;
82     }
83
85     virtual void setIrisProxyInterface(IrisProxyInterface* irisProxyInterface_) override
86     {
87         if (logMessages)
88         {
89             log.info("setIrisProxyInterface(irisProxyInterface=%p)\n", (void*)irisProxyInterface_);
90         }
91         irisProxyInterface = irisProxyInterface_ ? irisProxyInterface_ : &defaultIrisProxyInterface;
92     }
93
94 public:
95     // IrisInterface implementation.
```

```
96
98      virtual void  irisHandleMessage(const uint64_t* message) override;
99
100     // Set log level for logging messages.
101     void setLogLevel(unsigned level);
102
103     // Emit log message.
104     void emitLogMessage(const std::string& message, const std::string& severityLevel);
105
111     void setLogMessageFunction(std::function<IrisErrorCode(const std::string&, const std::string&)>
        func)
112     {
113         logMessageFunction = func;
114     }
115
116 private:
117     // --- Functions implemented locally in the global instance (registered in the functionDecoder). ---
118
120     void impl_instanceRegistry_registerInstance(IrisReceivedRequest& request);
121
123     void impl_instanceRegistry_unregisterInstance(IrisReceivedRequest& request);
124
126     void impl_instanceRegistry_getList(IrisReceivedRequest& request);
127
129     void impl_instanceRegistry_getInstanceInfoByInstId(IrisReceivedRequest& request);
130
132     void impl_instanceRegistry_getInstanceInfoByName(IrisReceivedRequest& request);
133
135     void impl_perInstanceExecution_setStateAll(IrisReceivedRequest& request);
136
138     void impl_perInstanceExecution_getStateAll(IrisReceivedRequest& request);
139
141     void impl_tcpServer_start(IrisReceivedRequest& request);
142
144     void impl_tcpServer_stop(IrisReceivedRequest& request);
145
147     void impl_tcpServer_getPort(IrisReceivedRequest& request);
148
150     void impl_plugin_load(IrisReceivedRequest& request);
151
153     void impl_service_connect(IrisReceivedRequest& request);
154
156     void impl_service_disconnect(IrisReceivedRequest& request);
157
159     void impl_logger_logMessage(IrisReceivedRequest& request);
160
161     // --- Private helpers ---
162
164     struct InstanceRegistryEntry
165     {
166         //             instId: The index in instanceRegistry is the instId.
167         std::string    instName;
168         uint64_t       channelId{IRIS_UINT64_MAX}; // If this is IRIS_UINT64_MAX this means this entry
        is unused.
169         IrisInterface* iris_interface{nullptr};
170         std::string     connectionInfo;
171
172         bool empty() const
173         {
174             return channelId == IRIS_UINT64_MAX;
175         }
176
178         void clear()
179         {
180             instName        = "";
181             channelId       = IRIS_UINT64_MAX;
182             iris_interface  = nullptr;
183             connectionInfo = "";
184
185             assert(empty());
186         }
187     };
188
190     InstanceId registerInstance(std::string&   instName,
191                                 uint64_t       channel_id,
192                                 bool           uniquify,
193                                 IrisInterface* iris_interface);
194
196     void unregisterInstanceAndGenerateEvent(InstanceRegistryEntry*  entry,
197                                             InstanceId              aInstId,
198                                             uint64_t                time,
199                                             std::list<IrisRequest>& deferred_event_requests);
200
202     const InstanceRegistryEntry* findInstanceRegistryEntry(InstanceId instId) const
203     {
204         if (instId >= InstanceId(instanceRegistry.size()))
205             return nullptr;
```

```
206
207         if (instanceRegistry[instId].empty())
208             return nullptr;
209
210         return &instanceRegistry[instId];
211     }
212
216     InstanceId addNewInstance(const std::string& instName,
217                               uint64_t           channelId,
218                               IrisInterface*     iris_interface);
219
222     TcpServerStartResult startServer(const std::string& connectionSpec);
223
224     // Stop the Iris Server (if running)
225     void stopServer();
226
227     // stop the Iris Client (if running)
228     void stopClient();
229
230     void loadPlugin(const std::string& plugin_path);
231
233     IrisErrorCode createEventStream(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&);
234
236     uint64_t getTimeForEvents();
237
240     std::string getInstName(InstanceId instId) const;
241
243     void initGlobalEventSources();
244
246     void registerGlobalFunctions();
247
248
249     // --- Private data ---
250
255     class Instance : public IrisInstance
256     {
257     public:
258         Instance()
259             : IrisInstance()
260         {
261             thisInstanceInfo.instName = "framework.GlobalInstance";
262             thisInstanceInfo.instId   = IrisInstIdGlobalInstance;
263             setProperty("instName", getInstanceName());
264             setProperty("instId", getInstId());
265             // NOTE: This instance does not think it is registered.
266             //       This means it won't unregister itself when it is destroyed but that doesn't matter.
267             //       We will be cleaning up all that state anyway.
268             log.setLogContext("IrisGI"); // Use a short prefix to keep log lines of all Iris calls
    short.
269         }
270
271         IrisInstanceEvent event_handler;
272     } irisInstance;
273
275     IrisEventRegistry instance_registry_changed_event_registry;
276
278     IrisEventRegistry shutdown_enter_event_registry;
279
281     IrisEventRegistry shutdown_leave_event_registry;
282
284     IrisEventRegistry log_message_event_registry;
285
288     std::vector<InstanceRegistryEntry> instanceRegistry;
289
291     std::mutex instance_registry_mutex;
292
295     std::vector<InstanceId> freeInstIds;
296
298     typedef std::map<std::string, uint64_t> InstanceRegistryNameToIdMap;
299
301     InstanceRegistryNameToIdMap instanceRegistryNameToId;
302
313     unsigned logMessages;
314
316     IrisLogger& log;
317
318     // TCP server. This won't start listening until startServer() is called.
319     impl::IrisTcpServer* tcp_server;
320
322     impl::IrisServiceClient* service_client;
323
324     // Create and manage communication channels
325     impl::IrisChannelRegistry channel_registry;
326
327     // --- Load and manage plugins ---
328     using Plugin = impl::IrisPlugin<IrisGlobalInstance>;
```

```
329     std::unordered_map<std::string, std::unique_ptr<Plugin» plugins;
330
331     std::mutex plugins_mutex;
332
333     std::mutex log_mutex;
334
339     class DefaultIrisProxyInterface : public IrisProxyInterface
340     {
341     public:
342         virtual void          irisHandleMessageInProxy(IrisInterface* irisInterface, InstanceId instId,
      const uint64_t* message) override;
343         virtual IrisErrorCode processAsyncMessagesInProxy(bool waitForAMessage) override;
344     } defaultIrisProxyInterface;
345
353     std::atomic<IrisProxyInterface*> irisProxyInterface{&defaultIrisProxyInterface};
354
356     std::function<IrisErrorCode(const std::string&, const std::string&)> logMessageFunction;
357 };
358
359 NAMESPACE_IRIS_END
360
361 #endif // #ifndef ARM_INCLUDE_IrisGlobalInstance_h
```

9.15 IrisInstance.h File Reference

Boilerplate code for an Iris instance, including clients and components.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisCppAdapter.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisFunctionDecoder.h"
#include "iris/detail/IrisObjects.h"
#include "iris/detail/IrisReceivedRequest.h"
#include "iris/IrisInstanceEvent.h"
#include <cassert>
#include <mutex>
#include <functional>
#include "iris/IrisInstanceBuilder.h"
```

Classes

- class iris::IrisInstance

Macros

- #define **irisRegisterEventBufferCallback**(instancePtr, instanceType, functionName, description) registerEventBufferCallback<instanceType, &instanceType::impl_##functionName>(instancePtr, #functionName, description, #instanceType)

  Register an event buffer callback function using an EventBufferCallbackDelegate.

- #define **irisRegisterEventCallback**(instancePtr, instanceType, functionName, description) registerEventCallback<instanceType, &instanceType::impl_##functionName>(instancePtr, #functionName, description, #instanceType)

  Register an event callback function using an EventCallbackDelegate Note: Use enableEvent() instead of irisRegisterEventCallback().

- #define **irisRegisterFunction**(instancePtr, instanceType, functionName, functionInfoJson) registerFunction(instancePtr, #functionName, &instanceType::impl_##functionName, functionInfoJson, #instanceType)

  Register an Iris function implementation. The function can be implemented in this class or in any other class. The helper macro is here to avoid repeating the function name. The 'impl_' prefix limits namespace pollution.

Typedefs

- typedef IrisDelegate< const EventBufferCallbackData & > **iris::EventBufferCallbackDelegate**

- typedef IrisDelegate< uint64_t, const IrisValueMap &, uint64_t, uint64_t, bool, std::string & > iris::EventCallbackDelegate

  Event callback delegate (deprecated)

9.15.1 Detailed Description

Boilerplate code for an Iris instance, including clients and components.

Copyright

Copyright (C) 2015-2024 Arm Limited. All rights reserved.

The IrisInstance class provides infrastructure that is:

- Necessary for all Iris instances.
- Useful for Iris components.
- Useful for Iris clients.

Note

Using this class to implement a correct Iris interface is optional. This class does not form an interface between instances. It just forms an interface between itself and the code of an instance.

This class is useful for, and used by, both components and clients.

9.15.2 Typedef Documentation

9.15.2.1 EventCallbackDelegate

```
typedef IrisDelegate<uint64_t, const IrisValueMap&, uint64_t, uint64_t, bool, std::string&>
iris::EventCallbackDelegate
```

Event callback delegate (deprecated)
Note: Use enableEvent() instead of irisRegisterEventCallback().
Used to register a function that can receive event callbacks.

```
iris::IrisErrorCode ec_FOO(EventStreamId esId, const iris::IrisValueMap &fields, uint64_t time,
                           InstanceId sInstId, bool syncEc, std::string &errorMessageOut)
```

Example:

```
class MyEventCallback
{
public:
    iris::IrisErrorCode impl_ec_FOO(EventStreamId esId, const iris::IrisValueMap &fields, uint64_t time,
                                    InstanceId sInstId, bool syncEc, std::string &errorMessageOut)
    {
        ...
        return E_ok;
    }
};
MyEventCallback* my_event_callback_ptr;
iris_instance->irisRegisterEventCallback(my_event_callback_ptr, MyEventCallback, ec_FOO, "Handle event
       FOO");
```

9.16 IrisInstance.h

Go to the documentation of this file.

```
1
19 #ifndef ARM_INCLUDE_IrisInstance_h
20 #define ARM_INCLUDE_IrisInstance_h
21
22 #include "iris/detail/IrisCommon.h"
23 #include "iris/detail/IrisCppAdapter.h"
24 #include "iris/detail/IrisDelegate.h"
25 #include "iris/detail/IrisFunctionDecoder.h"
26 #include "iris/detail/IrisObjects.h"
27 #include "iris/detail/IrisReceivedRequest.h"
28 #include "iris/IrisInstanceEvent.h"
```

```
29
30 #include <cassert>
31 #include <mutex>
32 #include <functional>
33
34 NAMESPACE_IRIS_START
35
66 typedef IrisDelegate<uint64_t, const IrisValueMap&, uint64_t, uint64_t, bool, std::string&>
       EventCallbackDelegate;
67 typedef IrisDelegate<const EventBufferCallbackData&> EventBufferCallbackDelegate;
68
69 class IrisInstantiationContext;
70 class IrisInstanceBuilder;
71
72 class IrisInstance
73 {
74 public:
75 // --- Construction and destruction. ---
76
83 #define irisRegisterFunction(instancePtr, instanceType, functionName, functionInfoJson)
       registerFunction(instancePtr, #functionName, &instanceType::impl_##functionName, functionInfoJson,
       #instanceType)
84
88 #define irisRegisterEventCallback(instancePtr, instanceType, functionName, description)
       registerEventCallback<instanceType, &instanceType::impl_##functionName>(instancePtr, #functionName,
       description, #instanceType)
89
91 #define irisRegisterEventBufferCallback(instancePtr, instanceType, functionName, description)
       registerEventBufferCallback<instanceType, &instanceType::impl_##functionName>(instancePtr,
       #functionName, description, #instanceType)
92
100     static const uint64_t UNIQUIFY = (1 << 0);
101
103     static const uint64_t THROW_ON_ERROR = (1 << 1);
104
106     static const uint64_t DEFAULT_FLAGS = THROW_ON_ERROR;
107
109     static const bool SYNCHRONOUS = true;
110
112     static const bool ASYNCHRONOUS = !SYNCHRONOUS;
113
133     IrisInstance(IrisConnectionInterface* connection_interface = nullptr,
134                  const std::string&       instName             = std::string(),
135                  uint64_t                 flags                = DEFAULT_FLAGS);
136
143     IrisInstance(IrisInstantiationContext* context);
144
146     ~IrisInstance();
147
155     void setConnectionInterface(IrisConnectionInterface* connection_interface);
156
161     void processAsyncRequests();
162
168     IrisInterface* getRemoteIrisInterface()
169     {
170         return remoteIrisInterface;
171     }
172
183     void setThrowOnError(bool throw_on_error)
184     {
185         default_cppAdapter = throw_on_error ? &throw_cppAdapter : &nothrow_cppAdapter;
186     }
187
201     IrisErrorCode registerInstance(const std::string& instName, uint64_t flags = DEFAULT_FLAGS);
202
208     IrisErrorCode unregisterInstance();
209
228     template <class T>
229     void setProperty(const std::string& propertyName, const T& propertyValue)
230     {
231         propertyMap[propertyName].set(propertyValue);
232     }
233
240     const PropertyMap& getPropertyMap() const
241     {
242         return propertyMap;
243     }
244
250     IrisLogger& getLogger()
251     {
252         return log;
253     }
254
255     // --- Interface for components. Provide functionality to clients. ---
256
269     template <class T>
270     void registerFunction(T* instance, const std::string& name, void
```

```
    (T::*memberFunctionPtr)(IrisReceivedRequest&), const std::string& functionInfoJson, const
    std::string& instanceTypeStr)
271     {
272         functionDecoder.registerFunction(instance, name, memberFunctionPtr, functionInfoJson,
    instanceTypeStr);
273     }
274
278     void unregisterFunction(const std::string& name)
279     {
280         functionDecoder.unregisterFunction(name);
281     }
282
294     template <class T>
295     void registerEventCallback(T* instance, const std::string& name, const std::string& description,
296                                void (T::*memberFunctionPtr)(IrisReceivedRequest&),
297                                const std::string& instanceTypeStr)
298     {
299         std::string funcInfoJson = "{description:'" + description +
300             "',"
301             "args:{"
302             "  instId:{type:'NumberU64', description:'Target instance id.'},"
303             "  esId:{type:'NumberU64', description:'Event stream id.'},"
304             "  fields:{type:'Object', description:'Object which contains the names and values of event
    source fields.'},"
305             "  time:{type:'NumberU64', description:'Simulation time timestamp of the event.'},"
306             "  sInstId:{type:'NumberU64', description:'Source instId: Instance which generated and sent
    this event.'},"
307             "  syncEc:{type:'Boolean', description:'Synchronous callback behaviour.', optional:true},"
308             "},"
309             "retval:{type:'Null'}}";
310         functionDecoder.registerFunction(instance, name, memberFunctionPtr, funcInfoJson,
    instanceTypeStr);
311     }
312
321     void registerEventCallback(EventCallbackDelegate delegate, const std::string& name,
322                                const std::string& description, const std::string& dlgInstanceTypeStr)
323     {
324         eventCallbacks[name] = ECD(delegate);
325         registerEventCallback(this, name, description, &IrisInstance::impl_eventCallback,
    dlgInstanceTypeStr);
326     }
327
336     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, const AttributeValueMap&, uint64_t,
    uint64_t, bool, std::string&)>
337     void registerEventCallback(T* instance, const std::string& name, const std::string& description,
338                                const std::string& dlgInstanceTypeStr)
339     {
340         registerEventCallback(EventCallbackDelegate::make<T, METHOD>(instance),
341                               name, description, dlgInstanceTypeStr);
342     }
343
355     template <class T>
356     void registerEventBufferCallback(T* instance, const std::string& name, const std::string&
    description,
357                                      void (T::*memberFunctionPtr)(IrisReceivedRequest&),
358                                      const std::string& instanceTypeStr)
359     {
360         std::string funcInfoJson = "{description:'" + description + "',"
361             "args:{"
362             "  instId:{type:'NumberU64', description:'Target instance id.'},"
363             "  sInstId:{type:'NumberU64', description:'Source instId: Instance which generated and sent
    this event buffer data.'},"
364             "  evBufId:{type:'NumberU64', description:'Event buffer id.'},"
365             "  events:{type:'EventData[]', description:'Array of EventData objects which represent the
    individual events in chronological order.'}"
366             "},"
367             "retval:{type:'Null'}}";
368         functionDecoder.registerFunction(instance, name, memberFunctionPtr, funcInfoJson,
    instanceTypeStr);
369     }
370
379     void registerEventBufferCallback(EventBufferCallbackDelegate delegate, const std::string& name,
380                                      const std::string& description, const std::string&
    dlgInstanceTypeStr)
381     {
382         eventBufferCallbacks[name] = EBCD(delegate);
383         registerEventBufferCallback(this, name, description, &IrisInstance::impl_eventBufferCallback,
    dlgInstanceTypeStr);
384     }
385
394     template <typename T, IrisErrorCode (T::*METHOD)(const EventBufferCallbackData& data)>
395     void registerEventBufferCallback(T* instance, const std::string& name, const std::string&
    description,
396                                      const std::string& dlgInstanceTypeStr)
397     {
398         registerEventBufferCallback(EventBufferCallbackDelegate::make<T, METHOD>(instance),
399                                     name, description, dlgInstanceTypeStr);
```

```
400     }
401
405     void unregisterEventCallback(const std::string& name);
406
410     void unregisterEventBufferCallback(const std::string& name);
411
422     using EventCallbackFunction = std::function<IrisErrorCode(EventStreamId, const IrisValueMap&,
    uint64_t, InstanceId, bool, std::string&)>;
423
427     void setCallback_IRIS_SIMULATION_TIME_EVENT(EventCallbackFunction f);
428
432     void setCallback_IRIS_SHUTDOWN_LEAVE(EventCallbackFunction f);
433
437     void addCallback_IRIS_INSTANCE_REGISTRY_CHANGED(EventCallbackFunction f);
438
446     void sendResponse(const uint64_t* response)
447     {
448         remoteIrisInterface->irisHandleMessage(response);
449     }
450
451     // --- Interface for clients. Access to other components. ---
452
458     IrisCppAdapter& irisCall() { return *default_cppAdapter; }
459
467     IrisCppAdapter& irisCallNoThrow() { return nothrow_cppAdapter; }
468
483     IrisCppAdapter& irisCallThrow() { return throw_cppAdapter; }
484
496     bool sendRequest(IrisRequest& req)
497     {
498         irisCall().callAndPerhapsWaitForResponse(req);
499         return req.hasOkResult();
500     }
501
502     // --- Misc functionality. ---
503
510     IrisInterface* getLocalIrisInterface() { return functionDecoder.getIrisInterface(); }
511
518     InstanceId getInstId() const { return thisInstanceInfo.instId; }
519
525     void setInstId(InstanceId instId) { thisInstanceInfo.instId = instId;
    cppAdapter_request_manager.setInstId(instId); }
526
536     const std::string& getInstanceName() const { return thisInstanceInfo.instName; }
537
540     bool isRegistered() const { return cppAdapter_request_manager.isRegistered(); }
541
548     IrisInstanceBuilder* getBuilder();
549
550     bool isAdapterInitialized() const { return is_adapter_initialized; }
551
552     void setAdapterInitialized() { is_adapter_initialized = true; }
553
557     void setEventHandler(IrisInstanceEvent* handler);
558
569     void notifyStateChanged();
570
581     template<class T>
582     void publishCppInterface(const std::string& interfaceName, T *pointer, const std::string&
    jsonDescription)
583     {
584         // Ignore null pointers: instance_getCppInterface...() promises to always return non-null
    pointers.
585         // (If there is no interface, do not publish it.)
586         if (pointer == nullptr)
587             return;
588
589         std::string functionInfoJson =
590             "{"
591             "    \"description\": \"" + jsonDescription + "\n"
592             "If this function is present it always returns a non-null pointer.\n"
593             "The caller of this function must make sure that the caller and callee use the same C++
    interface class layout and run in the same process. "
594             "This effectively means that they both must be compiled using the same compiler using the
    same header files. "
595             "The returned pointer is only meaningful if caller and callee run in the same process.\n"
596             "The meta-information provided alongside the returned pointer in CppInterfacePointer can
    (and should) be used to do minimal compatibility checking between caller and callee, see
    'CppInterfacePointer::isCompatibleWith()' in 'IrisObjects.h'.\","
597             "    \"args\": {"
598             "        \"instId\": {"
599             "            \"description\": \"Opaque number uniquely identifying the target instance.\","
600             "            \"type\": \"NumberU64\""
601             "        }"
602             "    },"
603             "    \"errors\": ["
604             "        \"E_unknown_instance_id\""
```

```
605             "    ],"
606             "    \"retval\": {"
607             "        \"description\": \"Pointer to the requested C++ interface (and associated
    meta-information) of this instance. Use 'CppInterfacePointer::isCompatibleWith()' to do a minimal
    compatibility check before using the pointer.\","
608             "        \"type\": \"CppInterfacePointer\""
609             "    }"
610             "}";
611         registerFunction(this, "instance_getCppInterface" + interfaceName,
    &IrisInstance::impl_instance_getCppInterface, functionInfoJson, "IrisInstance");
612         cppInterfaceRegistry[interfaceName].set(pointer);
613     }
614
623     void unpublishCppInterface(const std::string& interfaceName)
624     {
625         unregisterFunction("instance_getCppInterface" + interfaceName);
626         cppInterfaceRegistry.erase(interfaceName);
627     }
628
629     // --- Blocking simulation time functions ---
630
638     void simulationTimeRun();
639
645     void simulationTimeStop();
646
652     void simulationTimeRunUntilStop(double timeoutInSeconds = 0.0);
653
667     bool simulationTimeWaitForStop(double timeoutInSeconds = 0.0);
668
677     bool simulationTimeIsRunning();
678
691     void simulationTimeDisableEvents();
692
699     void setPendingSyncStepResponse(RequestId requestId);
700
706     bool setSyncStepEventBufferId(EventBufferId evBufId);
707
718     void eventBufferDestroyed(EventBufferId evBufId);
719
727     bool isValidEvBufId(EventBufferId evBufId) const;
728
772     std::vector<EventStreamInfo> findEventSourcesAndFields(const std::string& spec, InstanceId
    defaultInstId = IRIS_UINT64_MAX);
773     void findEventSourcesAndFields(const std::string& spec, std::vector<EventStreamInfo>&
    eventStreamInfosOut, InstanceId defaultInstId = IRIS_UINT64_MAX);
774
775
822     void enableEvent(const std::string& eventSpec, std::function<void (const EventStreamInfo&
    eventStreamInfo, IrisReceivedRequest& request)> callback, bool syncEc = ASYNCHRONOUS);
823
836     void enableEvent(const std::string& eventSpec, std::function<void ()> callback, bool syncEc =
    false);
837
856     void disableEvent(const std::string& eventSpec);
857
864     bool isEventEnabled(const std::string& eventSpec);
865
873     std::vector<InstanceInfo> findInstanceInfos(const std::string& instancePathFilter = "all");
874
881     std::vector<EventSourceInfo> findEventSources(const std::string& instancePathFilter = "all");
882
887     const std::vector<EventSourceInfo>& getEventSourceInfosOfAllInstances();
888
896     void destroyAllEventStreams();
897
905     const InstanceInfo& getInstanceInfo(InstanceId instId);
906
923     InstanceInfo getInstanceInfo(const std::string& instancePathFilter);
924
935     const std::vector<InstanceInfo>& getInstanceList();
936
946     std::string getInstanceName(InstanceId instId);
947
957     InstanceId getInstanceId(const std::string& instName);
958
968     ResourceId getResourceId(InstanceId instId, const std::string& resourceSpec);
969
994     uint64_t resourceRead(InstanceId instId, const std::string& resourceSpec);
995
1003     std::vector<uint64_t> resourceReadWide(InstanceId instId, const std::string& resourceSpec);
1004
1012     uint64_t resourceReadCrn(InstanceId instId, uint64_t canonicalRegisterNumber)
1013     {
1014         return resourceRead(instId, "crn:" + std::to_string(canonicalRegisterNumber));
1015     }
1016
1026     std::string resourceReadStr(InstanceId instId, const std::string& resourceSpec);
```

```
1027
1035     void resourceWrite(InstanceId instId, const std::string& resourceSpec, uint64_t value);
1036
1044     void resourceWrite(InstanceId instId, const std::string& resourceSpec, const std::vector<uint64_t>&
     value);
1045
1051     void resourceWriteCrn(InstanceId instId, uint64_t canonicalRegisterNumber, uint64_t value)
1052     {
1053         resourceWrite(instId, "crn:" + std::to_string(canonicalRegisterNumber), value);
1054     }
1055
1064     void resourceWriteStr(InstanceId instId, const std::string& resourceSpec, const std::string&
     value);
1065
1069     const std::vector<ResourceGroupInfo>& getResourceGroups(InstanceId instId);
1070
1074     const ResourceInfo& getResourceInfo(InstanceId instId, ResourceId resourceId);
1075
1081     const ResourceInfo& getResourceInfo(InstanceId instId, const std::string& resourceSpec);
1082
1090     std::vector<ResourceInfo> getResourceInfos(InstanceId instId, const std::string& resourceSpec);
1091
1095     const std::vector<ResourceInfo>& getResourceInfos(InstanceId instId);
1096
1100     MemorySpaceId getMemorySpaceId(InstanceId instId, uint64_t canonicalMsn);
1101
1110     MemorySpaceId getMemorySpaceId(InstanceId instId, const std::string& name);
1111
1115     const MemorySpaceInfo& getMemorySpaceInfo(InstanceId instId, uint64_t canonicalMsn);
1116
1120     const MemorySpaceInfo& getMemorySpaceInfoById(InstanceId instId, MemorySpaceId memorySpaceId);
1121
1133     const MemorySpaceInfo& getMemorySpaceInfo(InstanceId instId, const std::string& name);
1134
1138     const std::vector<MemorySpaceInfo>& getMemorySpaceInfos(InstanceId instId);
1139
1143     void clearCachedMetaInfo();
1144
1145 private:
1146     void init(IrisConnectionInterface* connection_interface_ = nullptr,
1147               const std::string&       instName              = std::string(),
1148               uint64_t                 flags                 = DEFAULT_FLAGS);
1149
1152     struct InstanceMetaInfo
1153     {
1156         std::map<std::string,ResourceId> resourceSpecToResourceIdAll;
1157
1161         std::map<std::string,ResourceId> resourceSpecToResourceIdUsed;
1162
1164         std::vector<ResourceGroupInfo> groupInfos;
1165
1167         std::vector<ResourceInfo> resourceInfos;
1168
1170         std::map<ResourceId,uint64_t> resourceIdToIndex;
1171
1173         std::vector<MemorySpaceInfo> memorySpaceInfos;
1174
1176         std::vector<EventSourceInfo> eventSourceInfos;
1177         bool eventSourceInfosValid{};
1178     };
1179
1183     InstanceMetaInfo& getInstanceMetaInfo(InstanceId instId);
1184
1188     IrisInstance::InstanceMetaInfo& getResourceMetaInfo(InstanceId instId);
1189
1193     IrisInstance::InstanceMetaInfo& getMemoryMetaInfo(InstanceId instId);
1194
1198     IrisInstance::InstanceMetaInfo& getEventSourceMetaInfo(InstanceId instId);
1199
1213     void expandWildcardsInEventStreamInfos(std::vector<EventStreamInfo>& eventStreamInfosInOut,
     InstanceId defaultInstId);
1214
1216     void enableSimulationTimeEvents();
1217
1219     void enableShutdownLeaveEvents();
1220
1222     void enableInstanceRegistryChangedEvent();
1223
1225     void simulationTimeWaitForRunning();
1226
1228     void simulationTimeClearGotRunning();
1229
1233     std::string lookupInstanceNameLocal(InstanceId instId);
1234
1236     void inFlightReceivedRequestsPush(IrisReceivedRequest *request)
1237     {
1238         assert(request);
```

```
1239         request->setNextInFlightReceivedRequest(inFlightReceivedRequestsHead);
1240         inFlightReceivedRequestsHead = request;
1241     }
1242
1244     IrisReceivedRequest *inFlightReceivedRequestsPop()
1245     {
1246         IrisReceivedRequest *r = inFlightReceivedRequestsHead;
1247         if (r)
1248         {
1249             inFlightReceivedRequestsHead = r->getNextInFlightReceivedRequest();
1250             r->setNextInFlightReceivedRequest(nullptr);
1251         }
1252         return r;
1253     }
1254
1255     // --- Iris function implementations ---
1256     void impl_instance_getProperties(IrisReceivedRequest& request);
1257     void impl_instance_ping(IrisReceivedRequest& request);
1258     void impl_instance_ping2(IrisReceivedRequest& request);
1259     void impl_batch_call(IrisReceivedRequest &request);
1260     void impl_instance_getCppInterface(IrisReceivedRequest& request);
1261
1263     void impl_eventCallback(IrisReceivedRequest& request);
1264
1266     void impl_eventBufferCallback(IrisReceivedRequest& request);
1267
1269     void impl_enableEventCallback(IrisReceivedRequest &request);
1270
1272     IrisErrorCode impl_ec_IrisInstance_IRIS_SIMULATION_TIME_EVENT(EventStreamId esId, const
      IrisValueMap& fields, uint64_t time,
1273                                                                   InstanceId sInstId, bool syncEc,
      std::string& errorMessageOut);
1274
1276     IrisErrorCode impl_ec_IrisInstance_IRIS_SHUTDOWN_LEAVE(EventStreamId esId, const IrisValueMap&
      fields, uint64_t time,
1277                                                            InstanceId sInstId, bool syncEc,
      std::string& errorMessageOut);
1278
1280     IrisErrorCode impl_ec_IrisInstance_IRIS_INSTANCE_REGISTRY_CHANGED(EventStreamId esId, const
      IrisValueMap& fields, uint64_t time,
1281                                                                       InstanceId sInstId, bool syncEc,
      std::string& errorMessageOut);
1282
1283     // --- Iris specific data and state ---
1284
1286     IrisFunctionDecoder functionDecoder{log, this};
1287
1289     IrisCppAdapter::RequestManager cppAdapter_request_manager{log};
1290
1292     IrisCppAdapter throw_cppAdapter{&cppAdapter_request_manager, /*throw_on_error=*/true};
1293
1295     IrisCppAdapter nothrow_cppAdapter{&cppAdapter_request_manager, /*throw_on_error=*/false};
1296
1298     IrisCppAdapter* default_cppAdapter{&throw_cppAdapter};
1299
1303     IrisConnectionInterface* connection_interface{nullptr};
1304
1307     IrisInterface* remoteIrisInterface{nullptr};
1308
1309 protected:
1311     InstanceInfo thisInstanceInfo{};
1312
1314     IrisLogger log;
1315
1316 private:
1318     bool instance_getProperties_called{false};
1319
1320     bool registered{false};
1321
1334     IrisReceivedRequest* inFlightReceivedRequestsHead{};
1335
1336     bool is_adapter_initialized{false};
1337
1338     uint64_t connectionInterfaceChannelId{IRIS_UINT64_MAX};
1339
1340     // --- Instance specific data and state ---
1341
1343     PropertyMap propertyMap{};
1344
1346     struct ECD
1347     {
1348         // Work around symbol length limits in Visual Studio (warning C4503)
1349         EventCallbackDelegate dlg;
1350         ECD() {}
1351         ECD(EventCallbackDelegate dlg_)
1352             : dlg(dlg_)
1353         {
```

```
1354         }
1355     };
1356     typedef std::map<std::string, ECD> EventCallbackMap;
1357     EventCallbackMap                   eventCallbacks{};
1358 
1360     struct EBCD
1361     {
1362         // Work around symbol length limits in Visual Studio (warning C4503)
1363         EventBufferCallbackDelegate dlg;
1364         EBCD() {}
1365         EBCD(EventBufferCallbackDelegate dlg_)
1366             : dlg(dlg_)
1367         {
1368         }
1369     };
1370     typedef std::map<std::string, EBCD> EventBufferCallbackMap;
1371     EventBufferCallbackMap              eventBufferCallbacks{};
1372 
1374     struct EnableEventCallbackInfo
1375     {
1376         EnableEventCallbackInfo() = default;
1377         EnableEventCallbackInfo(const EventStreamInfo& eventStreamInfo_, std::function<void (const
      EventStreamInfo& eventStreamInfo, IrisReceivedRequest& request)> callback_):
1378             eventStreamInfo(eventStreamInfo_),
1379             callback(callback_)
1380         {
1381         }
1382 
1383         EventStreamInfo eventStreamInfo;
1384         std::function<void (const EventStreamInfo& eventStreamInfo, IrisReceivedRequest& request)>
      callback;
1385     };
1386     typedef std::map<std::string, EnableEventCallbackInfo> EnableEventCallbackMap;
1387     EnableEventCallbackMap enableEventCallbacks;
1388 
1390     IrisInstanceBuilder* builder{nullptr};
1391 
1393     IrisInstanceEvent *irisInstanceEvent{};
1394 
1398     typedef std::map<std::string, CppInterfacePointer> CppInterfaceRegistryMap;
1399     CppInterfaceRegistryMap cppInterfaceRegistry{};
1400 
1402     bool simulationTimeIsRunning_{};
1403 
1405     bool simulationTimeGotRunningTrue{};
1406 
1408     bool simulationTimeGotRunningFalse{};
1409 
1411     std::mutex simulationTimeIsRunningMutex;
1412 
1414     std::condition_variable simulationTimeIsRunningChanged;
1415 
1417     EventStreamId simulationTimeEsId = IRIS_UINT64_MAX;
1418 
1420     EventStreamId shutdownLeaveEsId = IRIS_UINT64_MAX;
1421 
1423     EventStreamId instanceRegistryChangedEsId = IRIS_UINT64_MAX;
1424 
1426     EventCallbackFunction simulationTimeCallbackFunction;
1427 
1429     EventCallbackFunction shutdownLeaveCallbackFunction;
1430 
1431     // List of callback functions for IRIS_INSTANCE_REGISTRY_CHANGED.
1432     std::vector<EventCallbackFunction> instanceRegistryChangedFunctions;
1433 
1435     struct PendingSyncStepResponse
1436     {
1438         void setRequestId(RequestId requestId_)
1439         {
1440             requestId = requestId_;
1441         }
1442 
1444         void setEventBufferId(EventBufferId evBufId_)
1445         {
1446             evBufId = evBufId_;
1447         }
1448 
1450         bool isPending() const
1451         {
1452             return requestId != IRIS_UINT64_MAX;
1453         }
1454 
1456         void clear()
1457         {
1458             requestId = IRIS_UINT64_MAX;
1459         }
1460 
```

```
1462         void eventBufferDestroyed(EventBufferId evBufId_)
1463         {
1464             if (evBufId_ == evBufId)
1465             {
1466                 clear();
1467                 evBufId = IRIS_UINT64_MAX;
1468             }
1469         }
1470 
1473         RequestId requestId{IRIS_UINT64_MAX};
1474 
1476         EventBufferId evBufId{IRIS_UINT64_MAX};
1477     };
1478 
1480     PendingSyncStepResponse pendingSyncStepResponse;
1481 
1483 
1485     std::vector<InstanceInfo> instanceInfos;
1486 
1489     std::vector<uint64_t> instIdToIndex;
1490 
1492     std::map<InstanceId,InstanceMetaInfo> instIdToMetaInfo;
1493 
1495     std::vector<EventSourceInfo> eventSourceInfosOfAllInstances;
1496 };
1497 
1498 
1499 NAMESPACE_IRIS_END
1500 
1501 #endif // #ifndef ARM_INCLUDE_IrisInstance_h
1502 
1503 // Convenience #include.
1504 // (IrisInstanceBuilder needs the complete type of IrisInstance.)
1505 #include "iris/IrisInstanceBuilder.h"
1506 
```

9.17 IrisInstanceBreakpoint.h File Reference

Breakpoint add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include <cstdio>
#include <functional>
#include <list>
```

Classes

- struct iris::BreakpointHitInfo
- struct iris::BreakpointHitInfos
- class iris::IrisInstanceBreakpoint

  Breakpoint add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< const BreakpointInfo & > iris::BreakpointDeleteDelegate

  Delete the breakpoint corresponding to the given information.
- typedef IrisDelegate< BreakpointInfo & > iris::BreakpointSetDelegate

  Set a breakpoint corresponding to the given information.

9.17.1 Detailed Description

Breakpoint add-on to IrisInstance.

The IrisInstanceBreakpoint class:

- Implements all breakpoint-related Iris functions.
- Maintains and provides breakpoint information, for example type, address, and rscId.
- Converts between Iris breakpoint functions (breakpoint∗()) and various C++ access functions.

9.17.2 Typedef Documentation

9.17.2.1 BreakpointDeleteDelegate

`typedef IrisDelegate<const BreakpointInfo&> iris::BreakpointDeleteDelegate`
Delete the breakpoint corresponding to the given information.
`IrisErrorCode deleteBpt(const BreakpointInfo &bptInfo)`
The breakpoint is guaranteed to exist and to be valid.
Error: Return E_∗ error code if it failed to delete the breakpoint.

9.17.2.2 BreakpointSetDelegate

`typedef IrisDelegate<BreakpointInfo&> iris::BreakpointSetDelegate`
Set a breakpoint corresponding to the given information.
`IrisErrorCode setBpt(BreakpointInfo &bptInfo)`
The breakpoint information members are guaranteed to be valid. The BreakpointInfo is non-const as the metadata might need to be modified. For example, in some cases it might be useful to align the address and fix the size of a data breakpoint. It should never modify the bptId, which is uniquely set by this add-on.
Error: Return E_∗ error code if it failed to set the breakpoint.

9.18 IrisInstanceBreakpoint.h

Go to the documentation of this file.

```
1
12 #ifndef ARM_INCLUDE_IrisInstanceBreakpoint_h
13 #define ARM_INCLUDE_IrisInstanceBreakpoint_h
14
15 #include "iris/detail/IrisCommon.h"
16 #include "iris/detail/IrisDelegate.h"
17 #include "iris/detail/IrisLogger.h"
18 #include "iris/detail/IrisObjects.h"
19
20 #include <cstdio>
21 #include <functional>
22 #include <list>
23
24 NAMESPACE_IRIS_START
25
26 class IrisInstance;
27 class IrisInstanceEvent;
28 class IrisEventRegistry;
29 class IrisReceivedRequest;
30
31 class EventStream;
32 struct EventSourceInfo;
33
34 struct BreakpointHitInfo
35 {
36     //Required for all breakpoint types
37     const BreakpointInfo& bptInfo;
38
39     //Register and memory breakpoint
40     const std::vector<uint64_t> accessData;
41     const bool isReadAccess;
42 };
43
44 // we prefer a struct to an exposed vector so we can retain
45 // the API yet still modify struct sent to delegate in future
```

```
46 struct BreakpointHitInfos
47 {
48     std::vector<BreakpointHitInfo> breakpointHitInfos{};
49 };
50
64 typedef IrisDelegate<BreakpointInfo&> BreakpointSetDelegate;
65
76 typedef IrisDelegate<const BreakpointInfo&> BreakpointDeleteDelegate;
77
98 class IrisInstanceBreakpoint
99 {
100
101 public:
102     // --- Construction and destruction. ---
103     IrisInstanceBreakpoint(IrisInstance* irisInstance = nullptr);
104     ~IrisInstanceBreakpoint();
105
113     void attachTo(IrisInstance* irisInstance);
114
120     void setBreakpointSetDelegate(BreakpointSetDelegate delegate);
121
127     void setBreakpointDeleteDelegate(BreakpointDeleteDelegate delegate);
128
134     void setHandleBreakpointHitsDelegate(std::function<IrisErrorCode(const BreakpointHitInfos& hitBpts)>
    delegate);
135
141     void setEventHandler(IrisInstanceEvent* handler);
142
154     void notifyBreakpointHit(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId);
155
173     void notifyBreakpointHitData(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
174                                  uint64_t accessAddr, uint64_t accessSize,
175                                  const std::string& accessRw, const std::vector<uint64_t>& data);
176
192     void notifyBreakpointHitRegister(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
193                                      const std::string& accessRw, const std::vector<uint64_t>& data);
194
202     const BreakpointInfo* getBreakpointInfo(BreakpointId bptId) const;
203
213     void addCondition(const std::string& name, const std::string& type, const std::string& description,
214                       const std::vector<std::string> bpt_types = std::vector<std::string>());
215
222     void handleBreakpointHits(const BreakpointHitInfos& hitBpts);
223
224     bool hasAnyBreakpointSet() { return inUseBptIds.size() > 0; }
225
226 private:
227     void impl_breakpoint_set(IrisReceivedRequest& request);
228
229     void impl_breakpoint_delete(IrisReceivedRequest& request);
230
231     void impl_breakpoint_getList(IrisReceivedRequest& request);
232
233     void impl_breakpoint_getAdditionalConditions(IrisReceivedRequest& request);
234
235     bool validateInterceptionParameters(IrisReceivedRequest& request, const InterceptionParams&
    interceptionParams);
236
239     bool beginBreakpointHit(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId);
240
242     IrisErrorCode createEventStream(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&);
243
245     IrisErrorCode deleteBreakpoint(BreakpointId bpt);
246
247     void register_ec_IRIS_INSTANCE_REGISTRY_CHANGED();
248     IrisErrorCode ec_IRIS_INSTANCE_REGISTRY_CHANGED(EventStreamId esId, const IrisValueMap& fields,
    uint64_t time,
249                                                     InstanceId sInstId, bool syncEc, std::string&
    errorMessageOut);
250
252
254     IrisInstance* irisInstance;
255
257     IrisEventRegistry* breakpoint_hit_registry;
258
261     std::vector<BreakpointInfo> bptInfos;
262
265     std::vector<BreakpointId> freeBptIds;
266
269     std::list<BreakpointId> inUseBptIds;
270
272     std::map<BreakpointId, BreakpointAction> bptActions;
273
275     std::vector<BreakpointConditionInfo> additional_conditions;
```

```
276
278     BreakpointSetDelegate bptSetDelegate;
279
281     BreakpointDeleteDelegate bptDeleteDelegate;
282
284     std::function<IrisErrorCode(const BreakpointHitInfos& hitBpts)> handleBreakpointHitsDelegate;
285
287     IrisLogger log;
288
293     bool instance_registry_changed_registered{};
294 };
295
296 NAMESPACE_IRIS_END
297
298 #endif // #ifndef ARM_INCLUDE_IrisInstanceBreakpoint_h
```

9.19 IrisInstanceBuilder.h File Reference

A high level interface to build up functionality on an IrisInstance.

```
#include "iris/IrisEventEmitter.h"
#include "iris/IrisInstance.h"
#include "iris/IrisInstanceBreakpoint.h"
#include "iris/IrisInstanceDebuggableState.h"
#include "iris/IrisInstanceDisassembler.h"
#include "iris/IrisInstanceEvent.h"
#include "iris/IrisInstanceImage.h"
#include "iris/IrisInstanceMemory.h"
#include "iris/IrisInstancePerInstanceExecution.h"
#include "iris/IrisInstanceResource.h"
#include "iris/IrisInstanceSemihosting.h"
#include "iris/IrisInstanceCheckpoint.h"
#include "iris/IrisInstanceStep.h"
#include "iris/IrisInstanceTable.h"
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisElfDwarf.h"
#include <cassert>
#include <functional>
#include <utility>
```

Classes

- class iris::IrisInstanceBuilder::AddressTranslationBuilder

  Used to set metadata for an address translation.
- class iris::IrisInstanceBuilder::EventSourceBuilder

  Used to set metadata on an EventSource.
- class iris::IrisInstanceBuilder::FieldBuilder

  Used to set metadata on a register field resource.
- class iris::IrisInstanceBuilder

  Builder interface to populate an IrisInstance with registers, memory etc.
- class iris::IrisInstanceBuilder::MemorySpaceBuilder

  Used to set metadata for a memory space.
- class iris::IrisInstanceBuilder::ParameterBuilder

  Used to set metadata on a parameter.
- class iris::IrisInstanceBuilder::RegisterBuilder

  Used to set metadata on a register resource.
- struct iris::IrisInstanceBuilder::RegisterEventEmitterPair
- class iris::IrisInstanceBuilder::SemihostingManager

  semihosting_apis IrisInstanceBuilder semihosting APIs
- class iris::IrisInstanceBuilder::TableBuilder

Used to set metadata for a table.

- class iris::IrisInstanceBuilder::TableColumnBuilder

  Used to set metadata for a table column.

9.19.1 Detailed Description

A high level interface to build up functionality on an IrisInstance.

Copyright

9.20 IrisInstanceBuilder.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisInstanceBuilder_h
9 #define ARM_INCLUDE_IrisInstanceBuilder_h
10
11 #include "iris/IrisEventEmitter.h"
12 #include "iris/IrisInstance.h"
13 #include "iris/IrisInstanceBreakpoint.h"
14 #include "iris/IrisInstanceDebuggableState.h"
15 #include "iris/IrisInstanceDisassembler.h"
16 #include "iris/IrisInstanceEvent.h"
17 #include "iris/IrisInstanceImage.h"
18 #include "iris/IrisInstanceMemory.h"
19 #include "iris/IrisInstancePerInstanceExecution.h"
20 #include "iris/IrisInstanceResource.h"
21 #include "iris/IrisInstanceSemihosting.h"
22 #include "iris/IrisInstanceCheckpoint.h"
23 #include "iris/IrisInstanceStep.h"
24 #include "iris/IrisInstanceTable.h"
25 #include "iris/detail/IrisCommon.h"
26 #include "iris/detail/IrisElfDwarf.h"
27
28 #include <cassert>
29 #include <functional>
30 #include <utility>
31
32 NAMESPACE_IRIS_START
33
34 class IrisRegisterEventEmitterBase;
35
41 class IrisInstanceBuilder
42 {
43 private:
50     template <typename T, T* (IrisInstanceBuilder::*INIT_METHOD)()>
51     class LazyAddOn
52     {
53     private:
54         IrisInstanceBuilder* parent;
55         T*                   add_on;
56
57     public:
58         LazyAddOn(IrisInstanceBuilder* parent_)
59             : parent(parent_)
60             , add_on(nullptr)
61         {
62         }
63
64         ~LazyAddOn()
65         {
66             delete add_on;
67         }
68
69         T* operator->()
70         {
71             if (add_on == nullptr)
72             {
73                 init();
74             }
75
76             return add_on;
77         }
78
79         operator T*()
80         {
81             if (add_on == nullptr)
82             {
```

```
83                 init();
84             }
85
86             return add_on;
87         }
88
89         T* getPtr()
90         {
91             return add_on;
92         }
93
94         void init()
95         {
96             assert(add_on == nullptr);
97             add_on = (parent->*INIT_METHOD)();
98         }
99     };
100     IrisInstance* iris_instance;
101 #define INTERNAL_LAZY(addon) \
102     addon* init##addon();    \
103     LazyAddOn<addon, &IrisInstanceBuilder::init##addon>
104     INTERNAL_LAZY(IrisInstanceResource)
105     inst_resource;
106     INTERNAL_LAZY(IrisInstanceEvent)
107     inst_event;
108     INTERNAL_LAZY(IrisInstanceBreakpoint)
109     inst_breakpoint;
110     INTERNAL_LAZY(IrisInstanceMemory)
111     inst_memory;
112     INTERNAL_LAZY(IrisInstanceImage)
113     inst_image;
114     INTERNAL_LAZY(IrisInstanceImage_Callback)
115     inst_image_cb;
116     INTERNAL_LAZY(IrisInstanceStep)
117     inst_step;
118     INTERNAL_LAZY(IrisInstancePerInstanceExecution)
119     inst_per_inst_exec;
120     INTERNAL_LAZY(IrisInstanceTable)
121     inst_table;
122     INTERNAL_LAZY(IrisInstanceDisassembler)
123     inst_disass;
124     INTERNAL_LAZY(IrisInstanceDebuggableState)
125     inst_dbg_state;
126     INTERNAL_LAZY(IrisInstanceSemihosting)
127     inst_semihost;
128     INTERNAL_LAZY(IrisInstanceCheckpoint)
129     inst_checkpoint;
130 #undef INTERNAL_LAZY
131
133
134
136     ResourceReadDelegate  default_reg_read_delegate;
137     ResourceWriteDelegate default_reg_write_delegate;
138
141     bool canonicalRnSchemeIsAlreadySet{};
143
145
146     struct RegisterEventInfo
147     {
148         IrisInstanceEvent::EventSourceInfoAndDelegate event_info;
149
150         typedef std::vector<uint64_t> RscIdList;
151         RscIdList                     rscId_list;
152         IrisRegisterEventEmitterBase* event_emitter;
153
154         RegisterEventInfo()
155             : event_emitter(nullptr)
156         {
157         }
158     };
159
160     std::vector<RegisterEventInfo*> register_read_event_info_list;
161     std::vector<RegisterEventInfo*> register_update_event_info_list;
162
163     RegisterEventInfo* active_register_read_event_info{};
164     RegisterEventInfo* active_register_update_event_info{};
165
166     RegisterEventInfo* find_register_event(const std::vector<RegisterEventInfo*>&
    register_event_info_list,
167                                            const std::string&                     name);
168
169     RegisterEventInfo* initRegisterReadEventInfo(const std::string& name);
170     RegisterEventInfo* initRegisterUpdateEventInfo(const std::string& name);
171
174     void finalizeRegisterEvent(RegisterEventInfo* event_info, bool is_read);
176     std::string associateRegisterWithTraceEvents(ResourceId rscId);
178
```

```
180
181     IrisErrorCode setBreakpoint(BreakpointInfo& info);
182     IrisErrorCode deleteBreakpoint(const BreakpointInfo& info);
183
184     BreakpointSetDelegate    user_setBreakpoint;
185     BreakpointDeleteDelegate user_deleteBreakpoint;
187 public:
188     struct RegisterEventEmitterPair
189     {
190         IrisRegisterEventEmitterBase* read;
191         IrisRegisterEventEmitterBase* update;
192
193         RegisterEventEmitterPair()
194             : read(nullptr)
195             , update(nullptr)
196         {
197         }
198     };
199     typedef std::map<uint64_t, RegisterEventEmitterPair> RscIdEventEmitterMap;
200
201 private:
202     RscIdEventEmitterMap                                register_event_emitter_map;
203
204 public:
207     RscIdEventEmitterMap getRegisterEventEmitterMap()   {   return register_event_emitter_map;  }
208
214     IrisInstanceBuilder(IrisInstance* iris_instance);
215
216     /* No destructor: IrisInstanceBuilder objects live as long as the instance
217      * they belong to. Do not key anything to the destructor.
218      */
219
226 #define INTERNAL_RESOURCE_BUILDER_MIXIN(TYPE)
                \
227                                                                                 \
228                                                 \
229                     \
230     TYPE& setName(const std::string& name)
                \
231     {
                \
232         info->resourceInfo.name = name;
                \
233         return *this;
                \
234     }
                \
235                                                                                 \
236                                                 \
237                     \
238     TYPE& setCname(const std::string& cname)
                \
239     {
                \
240         info->resourceInfo.cname = cname;
                \
241         return *this;
                \
242     }
                \
243                                                                             \
244                                     \
245                     \
246     TYPE& setDescription(const std::string& description)
                \
247     {
                \
248         info->resourceInfo.description = description;
                \
249         return *this;
                \
250     }
                \
251                                                                 \
252     [[deprecated("Use setDescription() instead.")]]
                \
253     TYPE& setDescr(const std::string& description)
                \
254     {
                \
255         return setDescription(description);
                \
256     }
                \
257                                                                             \
258                                                 \
259                     \
```

```
260     TYPE& setFormat(const std::string& format)
                \
261     {
                \
262         info->resourceInfo.format = format;
                \
263         return *this;
                \
264     }
                \
265                                                                                 \
266                                           \
267                     \
268     TYPE& setBitWidth(uint64_t bitWidth)
                \
269     {
                \
270         info->resourceInfo.bitWidth = bitWidth;
                \
271         return *this;
                \
272     }
                \
273                                                                                     \
274                                                 \
275                     \
276     TYPE& setType(const std::string& type)
                \
277     {
                \
278         info->resourceInfo.type = type;
                \
279         return *this;
                \
280     }
                \
281                                                                                   \
282                                               \
283                     \
284     TYPE& setRwMode(const std::string& rwMode)
                \
285     {
                \
286         info->resourceInfo.rwMode = rwMode;
                \
287         return *this;
                \
288     }
                \
289                                                                                 \
290                                           \
291                     \
292     TYPE& setSubRscId(uint64_t subRscId)
                \
293     {
                \
294         info->resourceInfo.subRscId = subRscId;
                \
295         return *this;
                \
296     }
                \
297                                               \
298                                               \
299                               \
300                                                                 \
301                                                             \
302                     \
303     TYPE& addEnum(const std::string& symbol, const IrisValue& value, const std::string& description =
        std::string())      \
304     {
                \
305         info->resourceInfo.enums.push_back(EnumElementInfo(value, symbol, description));
                  \
306         return *this;
                \
307     }
                \
308                                               \
309                                               \
310                   \
311                                                             \
312                     \
313     TYPE& addStringEnum(const std::string& stringValue, const std::string& description = std::string())
                    \
314     {
                \
```

```
315         info->resourceInfo.enums.push_back(EnumElementInfo(IrisValue(stringValue), std::string(),
    description));          \
316         return *this;
                \
317     }
                \
318                                                         \
319                                                                 \
320                 \
321     TYPE& setTag(const std::string& tag)
                \
322     {
                \
323         info->resourceInfo.tags[tag] = IrisValue(true);
                \
324         return *this;
                \
325     }
            \
326                                                                     \
327                                                                   \
328                                                                  \
329                 \
330     TYPE& setTag(const std::string& tag, const IrisValue& value)
                \
331     {
                \
332         info->resourceInfo.tags[tag] = value;
                \
333         return *this;
                \
334     }
                \
335                                                                   \
336                                                               \
337                                                         \
338                                                           \
339                 \
340     TYPE& setReadDelegate(ResourceReadDelegate readDelegate)
                \
341     {
                \
342         info->readDelegate = readDelegate;
                \
343         return *this;
                \
344     }
                \
345                                                                 \
346                                                               \
347                                                         \
348                                                           \
349                 \
350     TYPE& setWriteDelegate(ResourceWriteDelegate writeDelegate)
                \
351     {
                \
352         info->writeDelegate = writeDelegate;
                \
353         return *this;
                \
354     }
                \
355                                                                   \
356                                             \
357                                                           \
358                                                     \
359   \
360                                               \
361                                         \
362                 \
363     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, ResourceReadResult&)>
                \
364     TYPE& setReadDelegate(T* instance)
                \
365     {
                \
366         return setReadDelegate(ResourceReadDelegate::make<T, METHOD>(instance));
                \
367     }
                \
368                                                                   \
369                                                                    \
370                                                               \
371                                                     \
372                                                         \
373                 \
```

```
374     template <IrisErrorCode (*FUNC)(const ResourceInfo&, ResourceReadResult&)>
                \
375     TYPE& setReadDelegate()
                \
376     {
                \
377         return setReadDelegate(ResourceReadDelegate::make<FUNC>());
                \
378     }
                \
379                                                             \
380                                             \
381                                                     \
382                                                 \
383 \
384                                               \
385                                         \
386                   \
387     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, const ResourceWriteValue&)>
                \
388     TYPE& setWriteDelegate(T* instance)
                \
389     {
                \
390         return setWriteDelegate(ResourceWriteDelegate::make<T, METHOD>(instance));
                \
391     }
                \
392                                                             \
393                                                               \
394                                                         \
395                                                 \
396                                                     \
397                   \
398     template <IrisErrorCode (*FUNC)(const ResourceInfo&, const ResourceWriteValue&)>
                \
399     TYPE& setWriteDelegate()
                \
400     {
                \
401         return setWriteDelegate(ResourceWriteDelegate::make<FUNC>());
                \
402     }
                \
403                                                                         \
404       \
405                                       \
406                                                 \
407                   \
408     TYPE& setParentRscId(ResourceId parentRscId)
                \
409     {
                \
410         info->resourceInfo.parentRscId = parentRscId;
                \
411         return *this;
                \
412     }
                \
413                                             \
414                                           \
415     ResourceId getRscId() const
                \
416     {
                \
417         return info->resourceInfo.rscId;
                \
418     }
                \
419                                                 \
420                           \
421                                                                 \
422                   \
423     TYPE& getRscId(ResourceId &rscIdOut)
                \
424     {
                \
425         rscIdOut = info->resourceInfo.rscId;
                \
426         return *this;
                \
427     }
428
429 #define INTERNAL_REGISTER_BUILDER_MIXIN(TYPE)
      \
430                                                             \
431                       \
```

```
432 \
433     TYPE& setLsbOffset(uint64_t lsbOffset)
    \
434     {
    \
435         info->resourceInfo.registerInfo.lsbOffset = lsbOffset;
    \
436         return *this;
    \
437     }
    \
438                                                         \
439                         \
440                     \
441 \
442     TYPE& setCanonicalRn(uint64_t canonicalRn_)
    \
443     {
    \
444         info->resourceInfo.registerInfo.canonicalRn    = canonicalRn_;
    \
445         info->resourceInfo.registerInfo.hasCanonicalRn = true;
    \
446         return *this;
    \
447     }
    \
448                                     \
449                                     \
450                                 \
451 \
452     TYPE& setCanonicalRnElfDwarf(uint16_t architecture, uint16_t dwarfRegNum)
    \
453     {
    \
454         if (!instance_builder->canonicalRnSchemeIsAlreadySet) /* Only set property if not already set.
    */\
455         {
    \
456             if (getWithDefault(instance_builder->iris_instance->getPropertyMap(),
    "register.canonicalRnScheme", "").getAsString().empty()) \
457             {
    \
458                 instance_builder->setPropertyCanonicalRnScheme("ElfDwarf");
    \
459             }
    \
460             instance_builder->canonicalRnSchemeIsAlreadySet = true;
    \
461         }
    \
462         return setCanonicalRn(makeCanonicalRnElfDwarf(architecture, dwarfRegNum));
    \
463     }
    \
464                                         \
465                     \
466             \
467                                         \
468 \
469     TYPE& setWriteMask(uint64_t value)
    \
470     {
    \
471         info->resourceInfo.setVector(info->resourceInfo.registerInfo.writeMask, value);
    \
472         return *this;
    \
473     }
    \
474                                           \
475         \
476       \
477                             \
478                     \
479             \
480                       \
481 \
482     template<typename Container>
    \
483     TYPE& setWriteMaskFromContainer(const Container& container)
    \
484     {
    \
485         info->resourceInfo.setVectorFromContainer(info->resourceInfo.registerInfo.writeMask, container);
    \
486         return *this;
```

```
    \
487     }
    \
488                                         \
489                 \
490                                                                 \
491                                           \
492                                               \
493 \
494     template<typename T>
    \
495     TYPE& setWriteMask(std::initializer_list<T>&& t)
    \
496     {
    \
497         setWriteMaskFromContainer(std::forward<std::initializer_list<T»(t));
    \
498         return *this;
    \
499     }
    \
500                                       \
501                   \
502           \
503                                       \
504 \
505     TYPE& setResetData(uint64_t value)
    \
506     {
    \
507         info->resourceInfo.setVector(info->resourceInfo.registerInfo.resetData, value);
    \
508         return *this;
    \
509     }
    \
510                                         \
511       \
512      \
513                         \
514                   \
515           \
516                     \
517 \
518     template<typename Container>
    \
519     TYPE& setResetDataFromContainer(const Container& container)
    \
520     {
    \
521         info->resourceInfo.setVectorFromContainer(info->resourceInfo.registerInfo.resetData, container);
    \
522         return *this;
    \
523     }
    \
524                                         \
525                 \
526                                                                 \
527                                           \
528                                               \
529 \
530     template<typename T>
    \
531     TYPE& setResetData(std::initializer_list<T>&& t)
    \
532     {
    \
533         setResetDataFromContainer(std::forward<std::initializer_list<T»(t));
    \
534         return *this;
    \
535     }
    \
536                                                       \
537                                                   \
538                     \
539 \
540     TYPE& setResetString(const std::string& resetString)
    \
541     {
    \
542         info->resourceInfo.registerInfo.resetString = resetString;
    \
543         return *this;
    \
544     }
```

```
    \
545                                                         \
546                     \
547 \
548     TYPE& setAddressOffset(uint64_t addressOffset)
    \
549     {
    \
550         info->resourceInfo.registerInfo.addressOffset    = addressOffset;
    \
551         info->resourceInfo.registerInfo.hasAddressOffset = true;
    \
552         return *this;
    \
553     }
    \
554                                                     \
555                     \
556      \
557     TYPE& setBreakpointSupportInfo(const std::string& supported)
    \
558     {
    \
559         info->resourceInfo.registerInfo.breakpointSupport = supported;
    \
560         return *this;
    \
561     }
562
563 #define INTERNAL_PARAMETER_BUILDER_MIXIN(TYPE)
    \
564                     \
565                     \
566           \
567                       \
568 \
569     TYPE& setDefaultData(uint64_t value)
    \
570     {
    \
571         info->resourceInfo.setVector(info->resourceInfo.parameterInfo.defaultData, value);
    \
572         return *this;
    \
573     }
    \
574                                   \
575         \
576       \
577                           \
578                     \
579           \
580                         \
581 \
582     template<typename Container>
    \
583     TYPE& setDefaultDataFromContainer(const Container& container)
    \
584     {
    \
585         info->resourceInfo.setVectorFromContainer(info->resourceInfo.parameterInfo.defaultData,
    container); \
586         return *this;
    \
587     }
    \
588                                   \
589                   \
590                                                           \
591                                             \
592                                                 \
593 \
594     template<typename T>
    \
595     TYPE& setDefaultData(std::initializer_list<T>&& t)
    \
596     {
    \
597         setDefaultDataFromContainer(std::forward<std::initializer_list<T»(t));
    \
598         return *this;
    \
599     }
    \
600           \
601                                                   \
602                     \
```

```
603 \
604     TYPE& setDefaultString(const std::string& defaultString)
    \
605     {
    \
606         info->resourceInfo.parameterInfo.defaultString = defaultString;
    \
607         return *this;
    \
608     }
    \
609                                                 \
610                                             \
611                                         \
612 \
613     TYPE& setInitOnly(bool initOnly = true)
    \
614     {
    \
615         info->resourceInfo.parameterInfo.initOnly = initOnly;
    \
616         /* Implicitly set read-only to make clear that parameter cannot be modified at run-time. */
    \
617         info->resourceInfo.rwMode = initOnly ? "r" : std::string(); /* =rw */
    \
618         return *this;
    \
619     }
    \
620                                                                 \
621                 \
622     /*                  but can still be accessed by resource_getResourceInfo() for clients that know the
    */\
623     /*                  resource name. */
    \
624           \
625     TYPE& setHidden(bool hidden = true)
    \
626     {
    \
627         info->resourceInfo.isHidden = hidden;
    \
628         return *this;
    \
629     }
    \
630                                               \
631                     \
632           \
633                                               \
634 \
635     TYPE& setMax(uint64_t value)
    \
636     {
    \
637         info->resourceInfo.setVector(info->resourceInfo.parameterInfo.max, value);
    \
638         return *this;
    \
639     }
    \
640                                         \
641       \
642       \
643                           \
644                     \
645           \
646                           \
647 \
648     template<typename Container>
    \
649     TYPE& setMaxFromContainer(const Container& container)
    \
650     {
    \
651         info->resourceInfo.setVectorFromContainer(info->resourceInfo.parameterInfo.max, container);
    \
652         return *this;
    \
653     }
    \
654                                         \
655                   \
656                                                                     \
657                                               \
658                                                   \
659 \
```

```
660     template<typename T>
    \
661     TYPE& setMax(std::initializer_list<T>&& t)
    \
662     {
    \
663         setMaxFromContainer(std::forward<std::initializer_list<T>>(t));
    \
664         return *this;
    \
665     }
    \
666                                             \
667                     \
668           \
669                                             \
670 \
671     TYPE& setMin(uint64_t value)
    \
672     {
    \
673         info->resourceInfo.setVector(info->resourceInfo.parameterInfo.min, value);
    \
674         return *this;
    \
675     }
    \
676                                         \
677       \
678      \
679                           \
680                     \
681           \
682                           \
683 \
684     template<typename Container>
    \
685     TYPE& setMinFromContainer(const Container& container)
    \
686     {
    \
687         info->resourceInfo.setVectorFromContainer(info->resourceInfo.parameterInfo.min, container);
    \
688         return *this;
    \
689     }
    \
690                                         \
691                   \
692                                                                       \
693                                             \
694                                                \
695 \
696     template<typename T>
    \
697     TYPE& setMin(std::initializer_list<T>&& t)
    \
698     {
    \
699         setMinFromContainer(std::forward<std::initializer_list<T>>(t));
    \
700         return *this;
    \
701     }
    \
702
706     class ParameterBuilder
707     {
708     private:
709         IrisInstanceResource::ResourceInfoAndAccess* info;
710
711     public:
712         ParameterBuilder(IrisInstanceResource::ResourceInfoAndAccess& info_)
713             : info(&info_)
714         {
715             info->resourceInfo.isParameter = true;
716         }
717
718         ParameterBuilder()
719             : info(nullptr)
720         {
721         }
722
723         INTERNAL_RESOURCE_BUILDER_MIXIN(ParameterBuilder)
724         INTERNAL_PARAMETER_BUILDER_MIXIN(ParameterBuilder)
725     };
726
```

```
727     class FieldBuilder;
728
732     class RegisterBuilder
733     {
734     private:
735         IrisInstanceResource::ResourceInfoAndAccess* info{};
736         IrisInstanceResource*                        inst_resource{};
737         IrisInstanceBuilder*                         instance_builder{};
738
739     public:
740         RegisterBuilder(IrisInstanceResource::ResourceInfoAndAccess& info_, IrisInstanceResource*
    inst_resource_, IrisInstanceBuilder *instance_builder_)
741             : info(&info_)
742             , inst_resource(inst_resource_)
743             , instance_builder(instance_builder_)
744         {
745             info->resourceInfo.isRegister = true;
746         }
747
748         RegisterBuilder()
749         {
750         }
751
752         INTERNAL_RESOURCE_BUILDER_MIXIN(RegisterBuilder)
753         INTERNAL_REGISTER_BUILDER_MIXIN(RegisterBuilder)
754
764         FieldBuilder addField(const std::string& name, uint64_t lsbOffset, uint64_t bitWidth, const
    std::string& description);
765
779         FieldBuilder addLogicalField(const std::string& name, uint64_t bitWidth, const std::string&
    description);
780     };
781
785     class FieldBuilder
786     {
787     protected:
788         IrisInstanceResource::ResourceInfoAndAccess* info{};
789         RegisterBuilder*                             parent_reg{};
790         IrisInstanceBuilder*                         instance_builder{};
791
792     public:
793         FieldBuilder(IrisInstanceResource::ResourceInfoAndAccess& info_, RegisterBuilder* parent_reg_,
    IrisInstanceBuilder *instance_builder_)
794             : info(&info_)
795             , parent_reg(parent_reg_)
796             , instance_builder(instance_builder_)
797         {
798         }
799
800         FieldBuilder()
801         {
802         }
803
804         INTERNAL_RESOURCE_BUILDER_MIXIN(FieldBuilder)
805         INTERNAL_REGISTER_BUILDER_MIXIN(FieldBuilder)
806
811         RegisterBuilder& parent()
812         {
813             return *parent_reg;
814         }
815
820         FieldBuilder addField(const std::string& name, uint64_t lsbOffset, uint64_t bitWidth, const
    std::string& description)
821         {
822             return parent().addField(name, lsbOffset, bitWidth, description);
823         }
824
829         FieldBuilder addLogicalField(const std::string& name, uint64_t bitWidth, const std::string&
    description)
830         {
831             return parent().addLogicalField(name, bitWidth, description);
832         }
833     };
834
835 #undef INTERNAL_RESOURCE_BUILDER_MIXIN
836 #undef INTERNAL_REGISTER_BUILDER_MIXIN
837 #undef INTERNAL_PARAMETER_BUILDER_MIXIN
838
869     void setDefaultResourceReadDelegate(ResourceReadDelegate delegate = ResourceReadDelegate())
870     {
871         default_reg_read_delegate = delegate;
872     }
873
901     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, ResourceReadResult&)>
902     void setDefaultResourceReadDelegate(T* instance)
903     {
904         setDefaultResourceReadDelegate(ResourceReadDelegate::make<T, METHOD>(instance));
```

```
905     }
906
926     template <IrisErrorCode (*FUNC)(const ResourceInfo&, ResourceReadResult&)>
927     void setDefaultResourceReadDelegate()
928     {
929         setDefaultResourceReadDelegate(ResourceReadDelegate::make<FUNC>());
930     }
931
961     void setDefaultResourceWriteDelegate(ResourceWriteDelegate delegate = ResourceWriteDelegate())
962     {
963         default_reg_write_delegate = delegate;
964     }
965
992     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, const ResourceWriteValue&)>
993     void setDefaultResourceWriteDelegate(T* instance)
994     {
995         setDefaultResourceWriteDelegate(ResourceWriteDelegate::make<T, METHOD>(instance));
996     }
997
1016     template <IrisErrorCode (*FUNC)(const ResourceInfo&, const ResourceWriteValue&)>
1017     void setDefaultResourceWriteDelegate()
1018     {
1019         setDefaultResourceWriteDelegate(ResourceWriteDelegate::make<*FUNC>());
1020     }
1021
1031     template <typename T, IrisErrorCode (T::*READER)(const ResourceInfo&, ResourceReadResult&),
1032               IrisErrorCode (T::*WRITER)(const ResourceInfo&, const ResourceWriteValue&)>
1033     void setDefaultResourceDelegates(T* instance)
1034     {
1035         setDefaultResourceReadDelegate(ResourceReadDelegate::make<T, READER>(instance));
1036         setDefaultResourceWriteDelegate(ResourceWriteDelegate::make<T, WRITER>(instance));
1037     }
1038
1061     void beginResourceGroup(const std::string& name,
1062                             const std::string& description,
1063                             uint64_t           subRscIdStart = IRIS_UINT64_MAX,
1064                             const std::string& cname         = std::string());
1065
1088     ParameterBuilder addParameter(const std::string& name, uint64_t bitWidth, const std::string&
    description);
1089
1108     ParameterBuilder addStringParameter(const std::string& name, const std::string& description);
1109
1143     RegisterBuilder addRegister(const std::string& name, uint64_t bitWidth, const std::string&
    description,
1144                                 uint64_t addressOffset = IRIS_UINT64_MAX, uint64_t canonicalRn =
    IRIS_UINT64_MAX);
1145
1164     RegisterBuilder addStringRegister(const std::string& name, const std::string& description);
1165
1186     RegisterBuilder addNoValueRegister(const std::string& name, const std::string& description, const
    std::string& format);
1187
1206     ParameterBuilder enhanceParameter(ResourceId rscId)
1207     {
1208         return ParameterBuilder(*(inst_resource->getResourceInfo(rscId)));
1209     }
1210
1232     RegisterBuilder enhanceRegister(ResourceId rscId)
1233     {
1234         return RegisterBuilder(*(inst_resource->getResourceInfo(rscId)), inst_resource, this);
1235     }
1236
1259     void setPropertyCanonicalRnScheme(const std::string& canonicalRnScheme);
1260
1268     void setNextSubRscId(uint64_t nextSubRscId)
1269     {
1270         inst_resource->setNextSubRscId(nextSubRscId);
1271     }
1272
1282     void setTag(ResourceId rscId, const std::string& tag);
1283
1291     const ResourceInfo &getResourceInfo(ResourceId rscId)
1292     {
1293         return inst_resource->getResourceInfo(rscId)->resourceInfo;
1294     }
1295
1296
1310     class EventSourceBuilder
1311     {
1312     private:
1313         IrisInstanceEvent::EventSourceInfoAndDelegate& info;
1314
1315     public:
1316         EventSourceBuilder(IrisInstanceEvent::EventSourceInfoAndDelegate& info_)
1317             : info(info_)
1318         {
```

```
1319         }
1320
1326         EventSourceBuilder& setName(const std::string& name)
1327         {
1328             info.info.name = name;
1329             return *this;
1330         }
1331
1337         EventSourceBuilder& setDescription(const std::string& description)
1338         {
1339             info.info.description = description;
1340             return *this;
1341         }
1342
1348         EventSourceBuilder& setFormat(const std::string& format)
1349         {
1350             info.info.format = format;
1351             return *this;
1352         }
1353
1359         EventSourceBuilder& setCounter(bool counter = true)
1360         {
1361             info.info.counter = counter;
1362             return *this;
1363         }
1364
1372         EventSourceBuilder& setHidden(bool hidden = true)
1373         {
1374             info.info.isHidden = hidden;
1375             return *this;
1376         }
1377
1384         EventSourceBuilder& hasSideEffects(bool hasSideEffects_ = true)
1385         {
1386             info.info.hasSideEffects = hasSideEffects_;
1387             return *this;
1388         }
1389
1402         EventSourceBuilder& addField(const std::string& name, const std::string& type, uint64_t
    sizeInBytes,
1403                                      const std::string& description)
1404         {
1405             info.info.addField(name, type, sizeInBytes, description);
1406             return *this;
1407         }
1408
1419         EventSourceBuilder& addEnumElement(uint64_t value, const std::string& symbol, const
    std::string& description = "")
1420         {
1421             if (info.info.fields.size() > 0)
1422             {
1423                 info.info.fields.back().addEnumElement(value, symbol, description);
1424                 return *this;
1425             }
1426             else
1427             {
1428                 throw IrisInternalError("EventSourceInfo has no fields to add an enum element to.");
1429             }
1430         }
1431
1441         EventSourceBuilder& addEnumElement(const std::string& fieldName, uint64_t value, const
    std::string& symbol, const std::string& description = "")
1442         {
1443             EventSourceFieldInfo *field = info.info.getField(fieldName);
1444             if (field == nullptr)
1445             {
1446                 throw IrisInternalError("addEnumElement(): Field " + fieldName + " not found");
1447             }
1448             field->addEnumElement(value, symbol, description);
1449             return *this;
1450         }
1451
1459         EventSourceBuilder& removeEnumElement(const std::string& fieldName, uint64_t value)
1460         {
1461             EventSourceFieldInfo *field = info.info.getField(fieldName);
1462             if (field == nullptr)
1463             {
1464                 throw IrisInternalError("removeEnumElement(): Field " + fieldName + " not found");
1465             }
1466             field->removeEnumElement(value);
1467             return *this;
1468         }
1469
1478         EventSourceBuilder& renameEnumElement(const std::string& fieldName, uint64_t value, const
    std::string& newEnumSymbol)
1479         {
1480             EventSourceFieldInfo *field = info.info.getField(fieldName);
```

```
1481            if (field == nullptr)
1482            {
1483                throw IrisInternalError("renameEnumElement(): Field " + fieldName + " not found");
1484            }
1485            field->renameEnumElement(value, newEnumSymbol);
1486            return *this;
1487        }
1488
1498        EventSourceBuilder& setEventStreamCreateDelegate(EventStreamCreateDelegate delegate)
1499        {
1500            info.createEventStream = delegate;
1501            return *this;
1502        }
1503
1516        template <typename T,
1517                  IrisErrorCode (T::*METHOD)(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&)>
1518        EventSourceBuilder& setEventStreamCreateDelegate(T* instance)
1519        {
1520            return setEventStreamCreateDelegate(EventStreamCreateDelegate::make<T, METHOD>(instance));
1521        }
1522
1536        template<typename T>
1537        EventSourceBuilder& addOption(const std::string& name, const std::string& type, const T&
    defaultValue,
1538                                      bool optional, const std::string& description)
1539        {
1540            info.info.addOption(name, type, defaultValue, optional, description);
1541            return *this;
1542        }
1543    };
1544
1559    EventSourceBuilder addEventSource(const std::string& name, bool isHidden = false)
1560    {
1561        return EventSourceBuilder(inst_event->addEventSource(name, isHidden));
1562    }
1563
1575    EventSourceBuilder addEventSource(const std::string& name, IrisEventEmitterBase& event_emitter,
    bool isHidden = false)
1576    {
1577        IrisInstanceEvent::EventSourceInfoAndDelegate& info = inst_event->addEventSource(name,
    isHidden);
1578
1579        event_emitter.setIrisInstance(iris_instance);
1580        event_emitter.setEvSrcId(info.info.evSrcId);
1581        info.createEventStream = EventStreamCreateDelegate::make<IrisEventEmitterBase,
1582
    &IrisEventEmitterBase::createEventStream>(&event_emitter);
1583
1584        return EventSourceBuilder(info);
1585    }
1586
1596    EventSourceBuilder enhanceEventSource(const std::string& name)
1597    {
1598        IrisInstanceEvent::EventSourceInfoAndDelegate& info = inst_event->enhanceEventSource(name);
1599        return EventSourceBuilder(info);
1600    }
1601
1608    void renameEventSource(const std::string& name, const std::string& newName)
1609    {
1610        inst_event->renameEventSource(name, newName);
1611    }
1612
1618    void deleteEventSource(const std::string& name)
1619    {
1620        inst_event->deleteEventSource(name);
1621    }
1622
1629    bool hasEventSource(const std::string& name)
1630    {
1631        return inst_event->hasEventSource(name);
1632    }
1633
1659    EventSourceBuilder setRegisterReadEvent(const std::string& name, const std::string& description =
    std::string());
1660
1686    EventSourceBuilder setRegisterReadEvent(const std::string& name, IrisRegisterEventEmitterBase&
    event_emitter);
1687
1694    void finalizeRegisterReadEvent();
1695
1722    EventSourceBuilder setRegisterUpdateEvent(const std::string& name, const std::string& description =
    std::string());
1723
1750    EventSourceBuilder setRegisterUpdateEvent(const std::string& name, IrisRegisterEventEmitterBase&
    event_emitter);
1751
```

```
1758     void finalizeRegisterUpdateEvent();
1759
1766     void resetRegisterReadEvent();
1767
1774     void resetRegisterUpdateEvent();
1775
1807     void setDefaultEsCreateDelegate(EventStreamCreateDelegate delegate)
1808     {
1809         inst_event->setDefaultEsCreateDelegate(delegate);
1810     }
1811
1842     template <typename T, IrisErrorCode (T::*METHOD)(EventStream*&, const EventSourceInfo&, const
     std::vector<std::string>&)>
1843     void setDefaultEsCreateDelegate(T* instance)
1844     {
1845         setDefaultEsCreateDelegate(EventStreamCreateDelegate::make<T, METHOD>(instance));
1846     }
1847
1870     template <IrisErrorCode (*FUNC)(EventStream*&, const EventSourceInfo&, const
     std::vector<std::string>&)>
1871     void setDefaultEsCreateDelegate()
1872     {
1873         setDefaultEsCreateDelegate(EventStreamCreateDelegate::make<FUNC>());
1874     }
1875
1882     IrisInstanceEvent* getIrisInstanceEvent() { return inst_event; }
1883
1915     void setBreakpointSetDelegate(BreakpointSetDelegate delegate)
1916     {
1917         if (inst_breakpoint.getPtr() == nullptr)
1918         {
1919             // Ensure the underlying IrisInstanceBreakpoint object is initialised too.
1920             inst_breakpoint.init();
1921         }
1922         user_setBreakpoint = delegate;
1923     }
1924
1946     template <typename T, IrisErrorCode (T::*METHOD)(BreakpointInfo&)>
1947     void setBreakpointSetDelegate(T* instance)
1948     {
1949         setBreakpointSetDelegate(BreakpointSetDelegate::make<T, METHOD>(instance));
1950     }
1951
1965     template <IrisErrorCode (*FUNC)(BreakpointInfo&)>
1966     void setBreakpointSetDelegate()
1967     {
1968         setBreakpointSetDelegate(BreakpointSetDelegate::make<FUNC>());
1969     }
1970
1992     void setBreakpointDeleteDelegate(BreakpointDeleteDelegate delegate)
1993     {
1994         if (inst_breakpoint.getPtr() == nullptr)
1995         {
1996             // Ensure the underlying IrisInstanceBreakpoint object is initialised too.
1997             inst_breakpoint.init();
1998         }
1999         user_deleteBreakpoint = delegate;
2000     }
2001
2023     template <typename T, IrisErrorCode (T::*METHOD)(const BreakpointInfo&)>
2024     void setBreakpointDeleteDelegate(T* instance)
2025     {
2026         setBreakpointDeleteDelegate(BreakpointDeleteDelegate::make<T, METHOD>(instance));
2027     }
2028
2042     template <IrisErrorCode (*FUNC)(const BreakpointInfo&)>
2043     void setBreakpointDeleteDelegate()
2044     {
2045         setBreakpointDeleteDelegate(BreakpointDeleteDelegate::make<FUNC>());
2046     }
2047
2068     void setHandleBreakpointHitsDelegate(std::function<IrisErrorCode(const BreakpointHitInfos&
     hitBpts)> delegate)
2069     {
2070         if (inst_breakpoint.getPtr() == nullptr)
2071         {
2072             // Ensure the underlying IrisInstanceBreakpoint object is initialised too.
2073             inst_breakpoint.init();
2074         }
2075
2076         inst_breakpoint->setHandleBreakpointHitsDelegate(std::move(delegate));
2077     }
2078
2089     void notifyBreakpointHit(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId)
2090     {
2091         inst_breakpoint->notifyBreakpointHit(bptId, time, pc, pcSpaceId);
2092     }
```

```
2093
2109      void notifyBreakpointHitData(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
2110                                   uint64_t accessAddr, uint64_t accessSize,
2111                                   const std::string& accessRw, const std::vector<uint64_t>& data)
2112      {
2113          inst_breakpoint->notifyBreakpointHitData(bptId, time, pc, pcSpaceId, accessAddr, accessSize,
    accessRw, data);
2114      }
2115
2129      void notifyBreakpointHitRegister(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
2130                                       const std::string& accessRw, const std::vector<uint64_t>& data)
2131      {
2132          inst_breakpoint->notifyBreakpointHitRegister(bptId, time, pc, pcSpaceId, accessRw, data);
2133      }
2134
2142      const BreakpointInfo* getBreakpointInfo(BreakpointId bptId)
2143      {
2144          return inst_breakpoint->getBreakpointInfo(bptId);
2145      }
2146
2148      void addBreakpointCondition(const std::string& name, const std::string& type, const std::string&
    description,
2149                                  const std::vector<std::string> bpt_types = std::vector<std::string>())
2150      {
2151          inst_breakpoint->addCondition(name, type, description, bpt_types);
2152      }
2153
2167      class MemorySpaceBuilder
2168      {
2169      private:
2170          IrisInstanceMemory::SpaceInfoAndAccess& info;
2171
2172      public:
2173          MemorySpaceBuilder(IrisInstanceMemory::SpaceInfoAndAccess& info_)
2174              : info(info_)
2175          {
2176          }
2177
2184          MemorySpaceBuilder& setName(const std::string& name)
2185          {
2186              info.spaceInfo.name = name;
2187              return *this;
2188          }
2189
2196          MemorySpaceBuilder& setDescription(const std::string& description)
2197          {
2198              info.spaceInfo.description = description;
2199              return *this;
2200          }
2201
2208          MemorySpaceBuilder& setMinAddr(uint64_t minAddr)
2209          {
2210              info.spaceInfo.minAddr = minAddr;
2211              return *this;
2212          }
2213
2220          MemorySpaceBuilder& setMaxAddr(uint64_t maxAddr)
2221          {
2222              info.spaceInfo.maxAddr = maxAddr;
2223              return *this;
2224          }
2225
2232          MemorySpaceBuilder& setCanonicalMsn(uint64_t canonicalMsn)
2233          {
2234              info.spaceInfo.canonicalMsn = canonicalMsn;
2235              return *this;
2236          }
2237
2244          MemorySpaceBuilder& setEndianness(const std::string& endianness)
2245          {
2246              info.spaceInfo.endianness = endianness;
2247              return *this;
2248          }
2249
2257          MemorySpaceBuilder& addAttribute(const std::string& name, AttributeInfo attrib)
2258          {
2259              info.spaceInfo.attrib[name] = attrib;
2260              return *this;
2261          }
2262
2269          MemorySpaceBuilder& setAttributes(const AttributeInfoMap& attribInfoMap)
2270          {
2271              info.spaceInfo.attrib = attribInfoMap;
2272              return *this;
2273          }
```

```
2274
2282        MemorySpaceBuilder& setAttributeDefault(const std::string& name, IrisValue value)
2283        {
2284            info.spaceInfo.attribDefaults[name] = value;
2285            return *this;
2286        }
2287
2300        MemorySpaceBuilder& setSupportedByteWidths(uint64_t supportedByteWidths)
2301        {
2302            info.spaceInfo.supportedByteWidths = supportedByteWidths;
2303            return *this;
2304        }
2305
2316        MemorySpaceBuilder& setReadDelegate(MemoryReadDelegate delegate)
2317        {
2318            info.readDelegate = delegate;
2319            return *this;
2320        }
2321
2332        MemorySpaceBuilder& setWriteDelegate(MemoryWriteDelegate delegate)
2333        {
2334            info.writeDelegate = delegate;
2335            return *this;
2336        }
2337
2348        MemorySpaceBuilder& setSidebandDelegate(MemoryGetSidebandInfoDelegate delegate)
2349        {
2350            info.sidebandDelegate = delegate;
2351            return *this;
2352        }
2353
2367        template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
    uint64_t, const AttributeValueMap&, MemoryReadResult&)>
2368        MemorySpaceBuilder& setReadDelegate(T* instance)
2369        {
2370            return setReadDelegate(MemoryReadDelegate::make<T, METHOD>(instance));
2371        }
2372
2386        template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
    uint64_t, const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2387        MemorySpaceBuilder& setWriteDelegate(T* instance)
2388        {
2389            return setWriteDelegate(MemoryWriteDelegate::make<T, METHOD>(instance));
2390        }
2391
2405        template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, const
    IrisValueMap&, const std::vector<std::string>&, IrisValueMap&)>
2406        MemorySpaceBuilder& setSidebandDelegate(T* instance)
2407        {
2408            return setSidebandDelegate(MemoryGetSidebandInfoDelegate::make<T, METHOD>(instance));
2409        }
2410
2421        template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2422                                        const AttributeValueMap&, MemoryReadResult&)>
2423        MemorySpaceBuilder& setReadDelegate()
2424        {
2425            return setReadDelegate(MemoryReadDelegate::make<FUNC>());
2426        }
2427
2438        template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2439                                        const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2440        MemorySpaceBuilder& setWriteDelegate()
2441        {
2442            return setWriteDelegate(MemoryWriteDelegate::make<FUNC>());
2443        }
2444
2455        template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, const IrisValueMap&,
2456                                        const std::vector<std::string>&, IrisValueMap&)>
2457        MemorySpaceBuilder& setSidebandDelegate()
2458        {
2459            return setSidebandDelegate(MemoryGetSidebandInfoDelegate::make<FUNC>());
2460        }
2461
2470        MemorySpaceId getSpaceId() const
2471        {
2472            return info.spaceInfo.spaceId;
2473        }
2474    };
2475
2479    class AddressTranslationBuilder
2480    {
2481    private:
2482        IrisInstanceMemory::AddressTranslationInfoAndAccess& info;
2483
2484    public:
2485        AddressTranslationBuilder(IrisInstanceMemory::AddressTranslationInfoAndAccess& info_)
2486            : info(info_)
```

```
2487         {
2488         }
2489
2500         AddressTranslationBuilder& setTranslateDelegate(MemoryAddressTranslateDelegate delegate)
2501         {
2502             info.translateDelegate = delegate;
2503             return *this;
2504         }
2505
2519         template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, uint64_t, uint64_t,
     MemoryAddressTranslationResult&)>
2520         AddressTranslationBuilder& setTranslateDelegate(T* instance)
2521         {
2522             return setTranslateDelegate(MemoryAddressTranslateDelegate::make<T, METHOD>(instance));
2523         }
2524
2535         template <IrisErrorCode (*FUNC)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult&)>
2536         AddressTranslationBuilder& setTranslateDelegate()
2537         {
2538             return setTranslateDelegate(MemoryAddressTranslateDelegate::make<FUNC>());
2539         }
2540     };
2541
2554     void setPropertyCanonicalMsnScheme(const std::string& canonicalMsnScheme);
2555
2588     void setDefaultMemoryReadDelegate(MemoryReadDelegate delegate = MemoryReadDelegate())
2589     {
2590         inst_memory->setDefaultReadDelegate(delegate);
2591     }
2592
2625     template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
     uint64_t, const AttributeValueMap&, MemoryReadResult&)>
2626     void setDefaultMemoryReadDelegate(T* instance)
2627     {
2628         setDefaultMemoryReadDelegate(MemoryReadDelegate::make<T, METHOD>(instance));
2629     }
2630
2656     template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2657                                     const AttributeValueMap&, MemoryReadResult&)>
2658     void setDefaultMemoryReadDelegate()
2659     {
2660         setDefaultMemoryReadDelegate(MemoryReadDelegate::make<FUNC>());
2661     }
2662
2696     void setDefaultMemoryWriteDelegate(MemoryWriteDelegate delegate = MemoryWriteDelegate())
2697     {
2698         inst_memory->setDefaultWriteDelegate(delegate);
2699     }
2700
2734     template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
     uint64_t, const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2735     void setDefaultMemoryWriteDelegate(T* instance)
2736     {
2737         setDefaultMemoryWriteDelegate(MemoryWriteDelegate::make<T, METHOD>(instance));
2738     }
2739
2765     template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2766                                     const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2767     void setDefaultMemoryWriteDelegate()
2768     {
2769         setDefaultMemoryWriteDelegate(MemoryWriteDelegate::make<FUNC>());
2770     }
2771
2790     MemorySpaceBuilder addMemorySpace(const std::string& name)
2791     {
2792         return MemorySpaceBuilder(inst_memory->addMemorySpace(name));
2793     }
2794
2826     void setDefaultAddressTranslateDelegate(MemoryAddressTranslateDelegate delegate =
     MemoryAddressTranslateDelegate())
2827     {
2828         inst_memory->setDefaultTranslateDelegate(delegate);
2829     }
2830
2858     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, uint64_t, uint64_t,
     MemoryAddressTranslationResult&)>
2859     void setDefaultAddressTranslateDelegate(T* instance)
2860     {
2861         setDefaultAddressTranslateDelegate(MemoryAddressTranslateDelegate::make<T, METHOD>(instance));
2862     }
2863
2883     template <IrisErrorCode (*FUNC)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult&)>
2884     void setDefaultAddressTranslateDelegate()
2885     {
2886         setDefaultAddressTranslateDelegate(MemoryAddressTranslateDelegate::make<FUNC>());
2887     }
2888
```

```
2905     AddressTranslationBuilder addAddressTranslation(MemorySpaceId inSpaceId, MemorySpaceId outSpaceId,
2906                                                     const std::string& description)
2907     {
2908         return AddressTranslationBuilder(inst_memory->addAddressTranslation(inSpaceId, outSpaceId,
     description));
2909     }
2910
2943     void setDefaultGetMemorySidebandInfoDelegate(MemoryGetSidebandInfoDelegate delegate)
2944     {
2945         inst_memory->setDefaultGetSidebandInfoDelegate(delegate);
2946     }
2947
2976     template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, const
     IrisValueMap&, const std::vector<std::string>&, IrisValueMap&)>
2977     void setDefaultGetMemorySidebandInfoDelegate(T* instance)
2978     {
2979         setDefaultGetMemorySidebandInfoDelegate(MemoryGetSidebandInfoDelegate::make<T,
     METHOD>(instance));
2980     }
2981
3002     template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, const IrisValueMap&,
3003                                     const std::vector<std::string>&, IrisValueMap&)>
3004     void setDefaultGetMemorySidebandInfoDelegate()
3005     {
3006         setDefaultGetMemorySidebandInfoDelegate(MemoryGetSidebandInfoDelegate::make<FUNC>());
3007     }
3008
3043     void setLoadImageFileDelegate(ImageLoadFileDelegate delegate = ImageLoadFileDelegate())
3044     {
3045         inst_image->setLoadImageFileDelegate(delegate);
3046     }
3047
3068     template <typename T, IrisErrorCode (T::*METHOD)(const std::string&)>
3069     void setLoadImageFileDelegate(T* instance)
3070     {
3071         setLoadImageFileDelegate(ImageLoadFileDelegate::make<T, METHOD>(instance));
3072     }
3073
3086     template <IrisErrorCode (*FUNC)(const std::string&)>
3087     void setLoadImageFileDelegate()
3088     {
3089         setLoadImageFileDelegate(ImageLoadFileDelegate::make<FUNC>());
3090     }
3091
3116     void setLoadImageDataDelegate(ImageLoadDataDelegate delegate = ImageLoadDataDelegate())
3117     {
3118         inst_image->setLoadImageDataDelegate(delegate);
3119     }
3120
3141     template <typename T, IrisErrorCode (T::*METHOD)(const std::vector<uint8_t>&)>
3142     void setLoadImageDataDelegate(T* instance)
3143     {
3144         setLoadImageDataDelegate(ImageLoadDataDelegate::make<T, METHOD>(instance));
3145     }
3146
3159     template <IrisErrorCode (*FUNC)(const std::vector<uint8_t>&)>
3160     void setLoadImageDataDelegate()
3161     {
3162         setLoadImageDataDelegate(ImageLoadDataDelegate::make<FUNC>());
3163     }
3164
3180     uint64_t openImage(const std::string& filename)
3181     {
3182         return inst_image_cb->openImage(filename);
3183     }
3184
3219     void setRemainingStepSetDelegate(RemainingStepSetDelegate delegate = RemainingStepSetDelegate())
3220     {
3221         inst_step->setRemainingStepSetDelegate(delegate);
3222     }
3223
3248     void setRemainingStepGetDelegate(RemainingStepGetDelegate delegate)
3249     {
3250         inst_step->setRemainingStepGetDelegate(delegate);
3251     }
3252
3273     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, const std::string&)>
3274     void setRemainingStepSetDelegate(T* instance)
3275     {
3276         setRemainingStepSetDelegate(RemainingStepSetDelegate::make<T, METHOD>(instance));
3277     }
3278
3299     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t&, const std::string&)>
3300     void setRemainingStepGetDelegate(T* instance)
3301     {
3302         setRemainingStepGetDelegate(RemainingStepGetDelegate::make<T, METHOD>(instance));
3303     }
```

```
3304
3317     template <IrisErrorCode (*FUNC)(uint64_t, const std::string&)>
3318     void setRemainingStepSetDelegate()
3319     {
3320         setRemainingStepSetDelegate(RemainingStepSetDelegate::make<FUNC>());
3321     }
3322
3335     template <IrisErrorCode (*FUNC)(uint64_t&, const std::string&)>
3336     void setRemainingStepGetDelegate()
3337     {
3338         setRemainingStepGetDelegate(RemainingStepGetDelegate::make<FUNC>());
3339     }
3340
3365     //
3366     void setStepCountGetDelegate(StepCountGetDelegate delegate = StepCountGetDelegate())
3367     {
3368         inst_step->setStepCountGetDelegate(delegate);
3369     }
3370
3391     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t&, const std::string&)>
3392     void setStepCountGetDelegate(T* instance)
3393     {
3394         setStepCountGetDelegate(RemainingStepGetDelegate::make<T, METHOD>(instance));
3395     }
3396
3409     template <IrisErrorCode (*FUNC)(uint64_t&, const std::string&)>
3410     void setStepCountGetDelegate()
3411     {
3412         setStepCountGetDelegate(RemainingStepGetDelegate::make<FUNC>());
3413     }
3414
3419     /*
3420      * @brief exec_apis IrisInstanceBuilder per-instance execution APIs
3421      * @{
3422      */
3423
3448     void setExecutionStateSetDelegate(PerInstanceExecutionStateSetDelegate delegate =
    PerInstanceExecutionStateSetDelegate())
3449     {
3450         inst_per_inst_exec->setExecutionStateSetDelegate(delegate);
3451     }
3452
3473     template <typename T, IrisErrorCode (T::*METHOD)(bool)>
3474     void setExecutionStateSetDelegate(T* instance)
3475     {
3476         setExecutionStateSetDelegate(PerInstanceExecutionStateSetDelegate::make<T, METHOD>(instance));
3477     }
3478
3491     template <IrisErrorCode (*FUNC)(bool)>
3492     void setExecutionStateSetDelegate()
3493     {
3494         setExecutionStateSetDelegate(PerInstanceExecutionStateSetDelegate::make<FUNC>());
3495     }
3496
3521     void setExecutionStateGetDelegate(PerInstanceExecutionStateGetDelegate delegate)
3522     {
3523         inst_per_inst_exec->setExecutionStateGetDelegate(delegate);
3524     }
3525
3546     template <typename T, IrisErrorCode (T::*METHOD)(bool&)>
3547     void setExecutionStateGetDelegate(T* instance)
3548     {
3549         setExecutionStateGetDelegate(PerInstanceExecutionStateGetDelegate::make<T, METHOD>(instance));
3550     }
3551
3564     template <IrisErrorCode (*FUNC)(bool&)>
3565     void setExecutionStateGetDelegate()
3566     {
3567         setExecutionStateGetDelegate(PerInstanceExecutionStateGetDelegate::make<FUNC>());
3568     }
3569
3574     /*
3575      * @brief table_apis IrisInstanceBuilder table APIs
3576      * @{
3577      */
3578
3579     class TableColumnBuilder;
3580
3584     class TableBuilder
3585     {
3586     private:
3587         IrisInstanceTable::TableInfoAndAccess& info;
3588
3589     public:
3590         TableBuilder(IrisInstanceTable::TableInfoAndAccess& info_)
3591             : info(info_)
3592         {
```

```
3593         }
3594
3600         TableBuilder& setName(const std::string& name)
3601         {
3602             info.tableInfo.name = name;
3603             return *this;
3604         }
3605
3611         TableBuilder& setDescription(const std::string& description)
3612         {
3613             info.tableInfo.description = description;
3614             return *this;
3615         }
3616
3622         TableBuilder& setMinIndex(uint64_t minIndex)
3623         {
3624             info.tableInfo.minIndex = minIndex;
3625             return *this;
3626         }
3627
3633         TableBuilder& setMaxIndex(uint64_t maxIndex)
3634         {
3635             info.tableInfo.maxIndex = maxIndex;
3636             return *this;
3637         }
3638
3644         TableBuilder& setIndexFormatHint(const std::string& hint)
3645         {
3646             info.tableInfo.indexFormatHint = hint;
3647             return *this;
3648         }
3649
3655         TableBuilder& setFormatShort(const std::string& format)
3656         {
3657             info.tableInfo.formatShort = format;
3658             return *this;
3659         }
3660
3666         TableBuilder& setFormatLong(const std::string& format)
3667         {
3668             info.tableInfo.formatLong = format;
3669             return *this;
3670         }
3671
3681         TableBuilder& setReadDelegate(TableReadDelegate delegate)
3682         {
3683             info.readDelegate = delegate;
3684             return *this;
3685         }
3686
3696         TableBuilder& setWriteDelegate(TableWriteDelegate delegate)
3697         {
3698             info.writeDelegate = delegate;
3699             return *this;
3700         }
3701
3713         template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, uint64_t, uint64_t,
       TableReadResult&)>
3714         TableBuilder& setReadDelegate(T* instance)
3715         {
3716             return setReadDelegate(TableReadDelegate::make<T, METHOD>(instance));
3717         }
3718
3730         template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, const TableRecords&,
       TableWriteResult&)>
3731         TableBuilder& setWriteDelegate(T* instance)
3732         {
3733             return setWriteDelegate(TableWriteDelegate::make<T, METHOD>(instance));
3734         }
3735
3745         template <IrisErrorCode (*FUNC)(const TableInfo&, uint64_t, uint64_t, TableReadResult&)>
3746         TableBuilder& setReadDelegate()
3747         {
3748             return setReadDelegate(TableReadDelegate::make<FUNC>());
3749         }
3750
3760         template <IrisErrorCode (*FUNC)(const TableInfo&, const TableRecords&, TableWriteResult&)>
3761         TableBuilder& setWriteDelegate()
3762         {
3763             return setWriteDelegate(TableWriteDelegate::make<FUNC>());
3764         }
3765
3776         TableBuilder& addColumnInfo(const TableColumnInfo& columnInfo)
3777         {
3778             info.tableInfo.columns.push_back(columnInfo);
3779             return *this;
3780         }
```

```
3781
3793        TableColumnBuilder addColumn(const std::string& name);
3794    };
3795
3799    class TableColumnBuilder
3800    {
3801    private:
3802        TableBuilder&    parent;
3803        TableColumnInfo& info;
3804
3805    public:
3806        TableColumnBuilder(TableBuilder& parent_, TableColumnInfo& info_)
3807            : parent(parent_)
3808            , info(info_)
3809        {
3810        }
3811
3821        TableBuilder& addColumnInfo(const TableColumnInfo& columnInfo)
3822        {
3823            return parent.addColumnInfo(columnInfo);
3824        }
3825
3837        TableColumnBuilder addColumn(const std::string& name) { return parent.addColumn(name); }
3838
3847        TableBuilder& endColumn()
3848        {
3849            return parent;
3850        }
3851
3858        TableColumnBuilder& setName(const std::string& name)
3859        {
3860            info.name = name;
3861            return *this;
3862        }
3863
3870        TableColumnBuilder& setDescription(const std::string& description)
3871        {
3872            info.description = description;
3873            return *this;
3874        }
3875
3882        TableColumnBuilder& setFormat(const std::string& format)
3883        {
3884            info.format = format;
3885            return *this;
3886        }
3887
3894        TableColumnBuilder& setType(const std::string& type)
3895        {
3896            info.type = type;
3897            return *this;
3898        }
3899
3906        TableColumnBuilder& setBitWidth(uint64_t bitWidth)
3907        {
3908            info.bitWidth = bitWidth;
3909            return *this;
3910        }
3911
3918        TableColumnBuilder& setFormatShort(const std::string& format)
3919        {
3920            info.formatShort = format;
3921            return *this;
3922        }
3923
3930        TableColumnBuilder& setFormatLong(const std::string& format)
3931        {
3932            info.formatLong = format;
3933            return *this;
3934        }
3935
3942        TableColumnBuilder& setRwMode(const std::string& rwMode)
3943        {
3944            info.rwMode = rwMode;
3945            return *this;
3946        }
3947    };
3948
3971    TableBuilder addTable(const std::string& name)
3972    {
3973        return TableBuilder(inst_table->addTableInfo(name));
3974    }
3975
4006    void setDefaultTableReadDelegate(TableReadDelegate delegate = TableReadDelegate())
4007    {
4008        inst_table->setDefaultReadDelegate(delegate);
4009    }
```

```
4010
4042     void setDefaultTableWriteDelegate(TableWriteDelegate delegate = TableWriteDelegate())
4043     {
4044         inst_table->setDefaultWriteDelegate(delegate);
4045     }
4046
4073     template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, uint64_t, uint64_t,
       TableReadResult&)>
4074     void setDefaultTableReadDelegate(T* instance)
4075     {
4076         setDefaultTableReadDelegate(TableReadDelegate::make<T, METHOD>(instance));
4077     }
4078
4106     template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, const TableRecords&,
       TableWriteResult&)>
4107     void setDefaultTableWriteDelegate(T* instance)
4108     {
4109         setDefaultTableWriteDelegate(TableWriteDelegate::make<T, METHOD>(instance));
4110     }
4111
4130     template <IrisErrorCode (*FUNC)(const TableInfo&, uint64_t, uint64_t, TableReadResult&)>
4131     void setDefaultTableReadDelegate()
4132     {
4133         setDefaultTableReadDelegate(TableReadDelegate::make<FUNC>());
4134     }
4135
4155     template <IrisErrorCode (*FUNC)(const TableInfo&, const TableRecords&, TableWriteResult&)>
4156     void setDefaultTableWriteDelegate()
4157     {
4158         setDefaultTableWriteDelegate(TableWriteDelegate::make<FUNC>());
4159     }
4160
4171     void setGetCurrentDisassemblyModeDelegate(GetCurrentDisassemblyModeDelegate delegate)
4172     {
4173         inst_disass->setGetCurrentModeDelegate(delegate);
4174     }
4175
4176     template <typename T, IrisErrorCode (T::*METHOD)(std::string&)>
4177     void setGetCurrentDisassemblyModeDelegate(T* instance)
4178     {
4179         setGetCurrentDisassemblyModeDelegate(GetCurrentDisassemblyModeDelegate::make<T,
       METHOD>(instance));
4180     }
4181
4183     void setGetDisassemblyDelegate(std::function<IrisErrorCode(GetDisassemblyArgs&)> delegate)
4184     {
4185         inst_disass->setGetDisassemblyDelegate(std::move(delegate));
4186     }
4187
4189     void setDisassembleOpcodeDelegate(DisassembleOpcodeDelegate delegate)
4190     {
4191         inst_disass->setDisassembleOpcodeDelegate(delegate);
4192     }
4193
4194     template <typename T, IrisErrorCode (T::*METHOD)(const std::vector<uint64_t>&, uint64_t, const
       std::string&, DisassembleContext&, DisassemblyLine&)>
4195     void setDisassembleOpcodeDelegate(T* instance)
4196     {
4197         setDisassembleOpcodeDelegate(DisassembleOpcodeDelegate::make<T, METHOD>(instance));
4198     }
4199
4200     template <IrisErrorCode (*FUNC)(const std::vector<uint64_t>&, uint64_t, const std::string&,
4201                                     DisassembleContext&, DisassemblyLine&)>
4202     void setDisassembleOpcodeDelegate()
4203     {
4204         setDisassembleOpcodeDelegate(DisassembleOpcodeDelegate::make<FUNC>());
4205     }
4206
4208     void addDisassemblyMode(const std::string& name, const std::string& description)
4209     {
4210         inst_disass->addDisassemblyMode(name, description);
4211     }
4212
4246     void setDbgStateSetRequestDelegate(DebuggableStateSetRequestDelegate delegate =
       DebuggableStateSetRequestDelegate())
4247     {
4248         inst_dbg_state->setSetRequestDelegate(delegate);
4249     }
4250
4271     template <typename T, IrisErrorCode (T::*METHOD)(bool)>
4272     void setDbgStateSetRequestDelegate(T* instance)
4273     {
4274         setDbgStateSetRequestDelegate(DebuggableStateSetRequestDelegate::make<T, METHOD>(instance));
4275     }
4276
4289     template <IrisErrorCode (*FUNC)(bool)>
4290     void setDbgStateSetRequestDelegate()
```

```
4291     {
4292         setDbgStateSetRequestDelegate(DebuggableStateSetRequestDelegate::make<FUNC>());
4293     }
4294
4319     void setDbgStateGetAcknowledgeDelegate(DebuggableStateGetAcknowledgeDelegate delegate =
     DebuggableStateGetAcknowledgeDelegate())
4320     {
4321         inst_dbg_state->setGetAcknowledgeDelegate(delegate);
4322     }
4323
4344     template <typename T, IrisErrorCode (T::*METHOD)(bool&)>
4345     void setDbgStateGetAcknowledgeDelegate(T* instance)
4346     {
4347         setDbgStateGetAcknowledgeDelegate(DebuggableStateGetAcknowledgeDelegate::make<T,
     METHOD>(instance));
4348     }
4349
4362     template <IrisErrorCode (*FUNC)(bool&)>
4363     void setDbgStateGetAcknowledgeDelegate()
4364     {
4365         setDbgStateGetAcknowledgeDelegate(DebuggableStateGetAcknowledgeDelegate::make<FUNC>());
4366     }
4367
4395     template <typename T, IrisErrorCode (T::*SET_REQUEST)(bool), IrisErrorCode
     (T::*GET_ACKNOWLEDGE)(bool&)>
4396     void setDbgStateDelegates(T* instance)
4397     {
4398         setDbgStateSetRequestDelegate<T, SET_REQUEST>(instance);
4399         setDbgStateGetAcknowledgeDelegate<T, GET_ACKNOWLEDGE>(instance);
4400     }
4401
4403     void setCheckpointSaveDelegate(CheckpointSaveDelegate delegate = CheckpointSaveDelegate())
4404     {
4405         inst_checkpoint->setCheckpointSaveDelegate(delegate);
4406     }
4407
4408     template <typename T, IrisErrorCode (T::*METHOD)(const std::string&)>
4409     void setCheckpointSaveDelegate(T* instance)
4410     {
4411         setCheckpointSaveDelegate(CheckpointSaveDelegate::make<T, METHOD>(instance));
4412     }
4413
4414     void setCheckpointRestoreDelegate(CheckpointRestoreDelegate delegate = CheckpointRestoreDelegate())
4415     {
4416         inst_checkpoint->setCheckpointRestoreDelegate(delegate);
4417     }
4418
4419     template <typename T, IrisErrorCode (T::*METHOD)(const std::string&)>
4420     void setCheckpointRestoreDelegate(T* instance)
4421     {
4422         setCheckpointRestoreDelegate(CheckpointRestoreDelegate::make<T, METHOD>(instance));
4423     }
4424
4437     class SemihostingManager
4438     {
4439     private:
4440         IrisInstanceSemihosting* inst_semihost;
4441
4442     public:
4443         SemihostingManager(IrisInstanceSemihosting* inst_semihost_)
4444             : inst_semihost(inst_semihost_)
4445         {
4446         }
4447
4448         ~SemihostingManager()
4449         {
4450             // Interrupt any requests that are currently blocked
4451             unblock();
4452         }
4453
4458         void enableExtensions()
4459         {
4460             inst_semihost->enableExtensions();
4461         }
4462
4477         std::vector<uint8_t> readData(uint64_t fDes, size_t max_size = 0, uint64_t flags =
     semihost::DEFAULT)
4478         {
4479             return inst_semihost->readData(fDes, max_size, flags);
4480         }
4481
4482         /*
4483          * @brief Write data for a given file descriptor
4484          *
4485          * @param   fDes        File descriptor to write to. Usually semihost::STDOUT or
     semihost::STDERR.
4486          * @param   data        Buffer containing the data to write.
```

```
4487          * @param   size        Size of the data buffer in bytes.
4488          * @return              Returns false if no client is registered for IRIS_SEMIHOSTING_OUTPUT
      events.
4489          */
4490         bool writeData(uint64_t fDes, const uint8_t* data, size_t size)
4491         {
4492             return inst_semihost->writeData(fDes, data, size);
4493         }
4494
4495         /*
4496          * @brief Write data for a given file descriptor
4497          *
4498          * @param   fDes        File descriptor to write to. Usually semihost::STDOUT or
      semihost::STDERR.
4499          * @param   data        Buffer containing the data to write.
4500          * @return              Returns false if no client is registered for IRIS_SEMIHOSTING_OUTPUT
      events.
4501          */
4502         bool writeData(uint64_t fDes, const std::vector<uint8_t>& data)
4503         {
4504             return writeData(fDes, &data.front(), data.size());
4505         }
4506
4521         std::pair<bool, uint64_t> semihostedCall(uint64_t operation, uint64_t parameter)
4522         {
4523             return inst_semihost->semihostedCall(operation, parameter);
4524         }
4525
4526         /*
4527          * @brief Request premature exit from any blocking requests that are currently blocked.
4528          */
4529         void unblock()
4530         {
4531             return inst_semihost->unblock();
4532         }
4533     };
4534
4542     SemihostingManager enableSemihostingAndGetManager()
4543     {
4544         inst_semihost.init();
4545         return SemihostingManager(inst_semihost);
4546     }
4547
4553     bool hasAnyBreakpointSetOrTraceEnabled()
4554     {
4555         if(inst_breakpoint && inst_breakpoint->hasAnyBreakpointSet())
4556             return true;
4557
4558         if(inst_event && inst_event->hasEventStreams())
4559             return true;
4560
4561         return false;
4562     }
4563
4567 };
4568
4569 inline IrisInstanceBuilder::TableColumnBuilder IrisInstanceBuilder::TableBuilder::addColumn(const
      std::string& name)
4570 {
4571     // Add a new column with default info
4572     info.tableInfo.columns.resize(info.tableInfo.columns.size() + 1);
4573     TableColumnInfo& col = info.tableInfo.columns.back();
4574
4575     col.name = name;
4576
4577     return TableColumnBuilder(*this, col);
4578 }
4579
4580 NAMESPACE_IRIS_END
4581
4582 #endif // ARM_INCLUDE_IrisInstanceBuilder_h
```

9.21 IrisInstanceCheckpoint.h File Reference

Checkpoint add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
```

Classes

- class iris::IrisInstanceCheckpoint

  Checkpoint add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< const std::string & > iris::CheckpointRestoreDelegate

  Restore the checkpoint corresponding to the given information.
- typedef IrisDelegate< const std::string & > iris::CheckpointSaveDelegate

  Save a checkpoint corresponding to the given information.

9.21.1 Detailed Description

Checkpoint add-on to IrisInstance.

Date

Copyright ARM Limited 2019 All Rights Reserved.

9.21.2 Typedef Documentation

9.21.2.1 CheckpointRestoreDelegate

```
typedef IrisDelegate<const std::string&> iris::CheckpointRestoreDelegate
```

Restore the checkpoint corresponding to the given information.

```
IrisErrorCode checkpoint_restore(const std::string & checkpoint_dir)
```

Error: Return E_* error code if it failed to restore the checkpoint.

9.21.2.2 CheckpointSaveDelegate

```
typedef IrisDelegate<const std::string&> iris::CheckpointSaveDelegate
```

Save a checkpoint corresponding to the given information.

```
IrisErrorCode checkpoint_save(const std::string & checkpoint_dir)
```

Error: Return E_* error code if it failed to save the checkpoint.

9.22 IrisInstanceCheckpoint.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisInstanceCheckpoint_h
8 #define ARM_INCLUDE_IrisInstanceCheckpoint_h
9
10 #include "iris/detail/IrisCommon.h"
11 #include "iris/detail/IrisDelegate.h"
12
13 NAMESPACE_IRIS_START
14
15 class IrisInstance;
16 class IrisReceivedRequest;
17
26 typedef IrisDelegate<const std::string&> CheckpointSaveDelegate;
27
36 typedef IrisDelegate<const std::string&> CheckpointRestoreDelegate;
37
41 class IrisInstanceCheckpoint
42 {
43
44 public:
45     IrisInstanceCheckpoint(IrisInstance* iris_instance = nullptr);
46
54     void attachTo(IrisInstance* iris_instance_);
55
61     void setCheckpointSaveDelegate(CheckpointSaveDelegate delegate);
62
68     void setCheckpointRestoreDelegate(CheckpointRestoreDelegate delegate);
```

```
69
70 private:
71     void impl_checkpoint_save(IrisReceivedRequest& request);
72
73     void impl_checkpoint_restore(IrisReceivedRequest& request);
74
76
78     IrisInstance* iris_instance;
79
81     CheckpointSaveDelegate save_delegate;
82
84     CheckpointRestoreDelegate restore_delegate;
85 };
86
87 NAMESPACE_IRIS_END
88
89 #endif // #ifndef ARM_INCLUDE_IrisInstanceCheckpoint_h
```

9.23 IrisInstanceDebuggableState.h File Reference

IrisInstance add-on to implement debuggableState functions.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
```

Classes

- class iris::IrisInstanceDebuggableState

  Debuggable-state add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< bool & > iris::DebuggableStateGetAcknowledgeDelegate

  Interface to stop the simulation time progress.
- typedef IrisDelegate< bool > iris::DebuggableStateSetRequestDelegate

  Delegate to set the debuggable-state-request flag.

9.23.1 Detailed Description

IrisInstance add-on to implement debuggableState functions.

Copyright

Copyright (C) 2017 Arm Limited. All rights reserved.

9.23.2 Typedef Documentation

9.23.2.1 DebuggableStateGetAcknowledgeDelegate

```
typedef IrisDelegate<bool&> iris::DebuggableStateGetAcknowledgeDelegate
```

Interface to stop the simulation time progress.

```
IrisErrorCode getAcknowledge(bool &acknowledge_out);
```

9.23.2.2 DebuggableStateSetRequestDelegate

```
typedef IrisDelegate<bool> iris::DebuggableStateSetRequestDelegate
```

Delegate to set the debuggable-state-request flag.

```
IrisErrorCode setRequest(bool request);
```

9.24 IrisInstanceDebuggableState.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisInstanceDebuggableState_h
9 #define ARM_INCLUDE_IrisInstanceDebuggableState_h
10
11 #include "iris/detail/IrisCommon.h"
12 #include "iris/detail/IrisDelegate.h"
13
14 NAMESPACE_IRIS_START
15
22 typedef IrisDelegate<bool> DebuggableStateSetRequestDelegate;
23
30 typedef IrisDelegate<bool&> DebuggableStateGetAcknowledgeDelegate;
31
32 class IrisInstance;
33 class IrisReceivedRequest;
34
38 class IrisInstanceDebuggableState
39 {
40 private:
42     IrisInstance* iris_instance;
43
45     DebuggableStateSetRequestDelegate     setRequest;
46     DebuggableStateGetAcknowledgeDelegate getAcknowledge;
47
48 public:
49     IrisInstanceDebuggableState(IrisInstance* iris_instance = nullptr);
50
56     void attachTo(IrisInstance* irisInstance);
57
65     void setSetRequestDelegate(DebuggableStateSetRequestDelegate delegate)
66     {
67         setRequest = delegate;
68     }
69
77     void setGetAcknowledgeDelegate(DebuggableStateGetAcknowledgeDelegate delegate)
78     {
79         getAcknowledge = delegate;
80     }
81
82 private:
84     void impl_debuggableState_setRequest(IrisReceivedRequest& request);
85
87     void impl_debuggableState_getAcknowledge(IrisReceivedRequest& request);
88 };
89
90 NAMESPACE_IRIS_END
91
92 #endif // ARM_INCLUDE_IrisInstanceSimulationTime_h
```

9.25 IrisInstanceDisassembler.h File Reference

Disassembler add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include <cstdio>
#include <utility>
```

Classes

- struct iris::GetDisassemblyArgs
- class iris::IrisInstanceDisassembler

  Disassembler add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< const std::vector< uint64_t > &, uint64_t, const std::string &, DisassembleContext &, DisassemblyLine & > iris::DisassembleOpcodeDelegate

Get the disassembly for an individual opcode.

- typedef IrisDelegate< std::string & > iris::GetCurrentDisassemblyModeDelegate

  Get the current disassembly mode.

9.25.1 Detailed Description

Disassembler add-on to IrisInstance.

Copyright

The IrisInstanceDisassembler class implements all disassembly-related Iris functions.

9.26 IrisInstanceDisassembler.h

Go to the documentation of this file.

```
1
9 #ifndef ARM_INCLUDE_IrisInstanceDisassembler_h
10 #define ARM_INCLUDE_IrisInstanceDisassembler_h
11
12 #include "iris/detail/IrisCommon.h"
13 #include "iris/detail/IrisDelegate.h"
14 #include "iris/detail/IrisLogger.h"
15 #include "iris/detail/IrisObjects.h"
16
17 #include <cstdio>
18 #include <utility>
19
20 NAMESPACE_IRIS_START
21
22 class IrisInstance;
23 class IrisReceivedRequest;
24
25 struct GetDisassemblyArgs
26 {
27 // Input args:
28     MemorySpaceId spaceId;
29     uint64_t address;
30     std::string mode;
31     uint64_t count;
32     uint64_t maxAddr;
33     AttributeValueMap attrib;
34 // Output args:
35     std::vector<DisassemblyLine> &disassemblyLineOut;
36 };
37
53 typedef IrisDelegate<std::string&> GetCurrentDisassemblyModeDelegate;
54
65 typedef IrisDelegate<const std::vector<uint64_t>&, uint64_t, const std::string&,
66                      DisassembleContext&, DisassemblyLine&>
67     DisassembleOpcodeDelegate;
68
69 /*
70  * @}
71  */
72
90 class IrisInstanceDisassembler
91 {
92 public:
98     IrisInstanceDisassembler(IrisInstance* irisInstance = nullptr);
99
105     void attachTo(IrisInstance* irisInstance);
106
114     void setGetCurrentModeDelegate(GetCurrentDisassemblyModeDelegate delegate)
115     {
116         getCurrentMode = delegate;
117     }
118
126     void setGetDisassemblyDelegate(std::function<IrisErrorCode(GetDisassemblyArgs&)> delegate)
127     {
128         getDisassembly = std::move(delegate);
129     }
130
138     void setDisassembleOpcodeDelegate(DisassembleOpcodeDelegate delegate)
139     {
140         disassembleOpcode = delegate;
141     }
142
```

```
152     void addDisassemblyMode(const std::string& name, const std::string& description);
153
154 private:
155     void impl_disassembler_getModes(IrisReceivedRequest& request);
156
157     void impl_disassembler_getCurrentMode(IrisReceivedRequest& request);
158
159     void impl_disassembler_getDisassembly(IrisReceivedRequest& request);
160
161     void impl_disassembler_disassembleOpcode(IrisReceivedRequest& request);
162
165     void checkDisassemblyMode(std::string& mode, bool& isValidMode);
166
168
170     IrisInstance* irisInstance;
171
173     GetCurrentDisassemblyModeDelegate getCurrentMode;
174
176     std::function<IrisErrorCode(GetDisassemblyArgs&)> getDisassembly;
177
179     DisassembleOpcodeDelegate disassembleOpcode;
180
181     std::vector<DisassemblyMode> disassemblyModes;
183     IrisLogger log;
184 };
185
186 NAMESPACE_IRIS_END
187
188 #endif // #ifndef ARM_INCLUDE_IrisInstanceDisassembler_h
```

9.27 IrisInstanceEvent.h File Reference

Event add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include "iris/detail/IrisRequest.h"
#include <cstdio>
#include <set>
#include <list>
```

Classes

- struct iris::IrisInstanceEvent::EventSourceInfoAndDelegate

  Contains the metadata and delegates for a single EventSource.
- class iris::EventStream

  Base class for event streams.
- class iris::IrisEventRegistry

  Class to register Iris event streams for an event.
- class iris::IrisEventStream

  Event stream class for Iris-specific events.
- class iris::IrisInstanceEvent

  Event add-on for IrisInstance.
- struct iris::IrisInstanceEvent::ProxyEventInfo

  Contains information for a single proxy EventSource.

Typedefs

- typedef IrisDelegate< EventStream *&, const EventSourceInfo &, const std::vector< std::string > & > iris::EventStreamCreateDelegate

  Delegate to create an EventStream.

9.27.1 Detailed Description

Event add-on to IrisInstance.

Copyright

Copyright (C) 2016-2024 Arm Limited. All rights reserved.

The IrisInstanceEvent class:

- Implements all event-related Iris functions.
- Maintains and provides event source metadata.
- Converts between Iris event functions (event∗()) and various C++ access functions.

9.27.2 Typedef Documentation

9.27.2.1 EventStreamCreateDelegate

```
typedef IrisDelegate<EventStream*&, const EventSourceInfo&, const std::vector<std::string>&>
iris::EventStreamCreateDelegate
```

Delegate to create an EventStream.

```
IrisErrorCode create(EventStream *&evStream, const EventSourceInfo &srcInfo, const std::vector<std::string>
    &fields)
```

Create a new event stream with the specified fields for an event source.
The new event stream is maintained and destroyed in the event add-on.
Error: Return E_∗ error code, for example E_unknown_event_field, if the event stream could not be created.

9.28 IrisInstanceEvent.h

Go to the documentation of this file.

```
1
12 #ifndef ARM_INCLUDE_IrisInstanceEvent_h
13 #define ARM_INCLUDE_IrisInstanceEvent_h
14
15 #include "iris/detail/IrisCommon.h"
16 #include "iris/detail/IrisDelegate.h"
17 #include "iris/detail/IrisLogger.h"
18 #include "iris/detail/IrisObjects.h"
19 #include "iris/detail/IrisRequest.h"
20
21 #include <cstdio>
22 #include <set>
23 #include <list>
24
25 NAMESPACE_IRIS_START
26
27 class IrisInstance;
28 class IrisReceivedRequest;
29
30 class EventStream;
31 class IrisEventRegistry;
32
45 typedef IrisDelegate<EventStream*&, const EventSourceInfo&, const std::vector<std::string>&>
    EventStreamCreateDelegate;
46
64 class IrisInstanceEvent
65 {
66 public:
67
68     /* ! What is a proxy event source?
69        - The event source in actual does not belong to this Iris instance, but instead belongs to another
    Iris instance (target).
70        - The event source is registered as a proxy in this Iris instance using Iris interface -
    event_registerProxyEventSource()
71        - This Iris instance acts as a proxy for those registered events.
72        - All interface calls (for example, eventStream_create) on the proxy event source are forwarded to
    the target instance.
73        - Similarly, all the created event streams in this Iris instance for the proxy event source are
    tagged as proxyForOtherInstance
74        - All the interface calls (for example, eventStream_enable) on such proxy event streams are
    forwarded to the target instance.
```

```
75        - Finally, the proxy event source can be deregistered using Iris interface -
      event_unregisterProxyEventSource()
76      */
77
81      struct ProxyEventInfo
82      {
83          InstanceId targetInstId{};      //target Iris instance Id
84          EventSourceId targetEvSrcId{};  //event source ID in target Iris instance
85          std::vector<EventStreamId> evStreamIds; //list of created event stream IDs
86          //Important note: When we create an event stream, we use the same esID for both - this and target
      Iris instance
87      };
88
92      struct EventSourceInfoAndDelegate
93      {
94          EventSourceInfo           info;
95          EventStreamCreateDelegate createEventStream;
96
97          bool isValid{true}; //deleteEventSource() sets isValid to false
98          bool isProxy{false};
99          ProxyEventInfo proxyEventInfo; //contains proper values only if isProxy=true
100     };
101
107     IrisInstanceEvent(IrisInstance* irisInstance = nullptr);
108     ~IrisInstanceEvent();
109
117     void attachTo(IrisInstance* irisInstance);
118
126     void setDefaultEsCreateDelegate(EventStreamCreateDelegate delegate);
127
140     EventSourceInfoAndDelegate& addEventSource(const std::string& name, bool isHidden = false);
141
149     uint64_t addEventSource(const EventSourceInfoAndDelegate& info);
150
159     EventSourceInfoAndDelegate& enhanceEventSource(const std::string& name);
160
169     void renameEventSource(const std::string& name, const std::string& newName);
170
176     void deleteEventSource(const std::string& eventName);
177
184     bool hasEventSource(const std::string& eventName);
185
193     const uint64_t *eventBufferGetSyncStepResponse(EventBufferId evBufId, RequestId requestId);
194
203     void eventBufferClear(EventBufferId evBufId);
204
212     bool isValidEvBufId(EventBufferId evBufId) const;
213
214
223     bool destroyEventStream(EventStreamId esId);
224
235     void destroyAllEventStreams();
236
242     const EventSourceInfo *getEventSourceInfo(EventSourceId evSrcId) const;
243
244 private:
245     // --- Iris function implementations ---
246
247     void impl_event_getEventSources(IrisReceivedRequest& request);
248
249     void impl_event_getEventSource(IrisReceivedRequest& request);
250
251     void impl_eventStream_create(IrisReceivedRequest& request);
252
253     void impl_eventStream_destroy(IrisReceivedRequest& request);
254
255     void impl_eventStream_destroyAll(IrisReceivedRequest& request);
256
257     void impl_eventStream_enable(IrisReceivedRequest& request);
258
259     void impl_eventStream_disable(IrisReceivedRequest& request);
260
261     void impl_eventStream_getCounter(IrisReceivedRequest& request);
262
263     void impl_eventStream_setTraceRanges(IrisReceivedRequest& request);
264
265     void impl_eventStream_getState(IrisReceivedRequest& request);
266
267     void impl_eventStream_flush(IrisReceivedRequest& request);
268
269     void impl_eventStream_setOptions(IrisReceivedRequest& request);
270
271     void impl_eventStream_action(IrisReceivedRequest& request);
272
273     void impl_eventBuffer_create(IrisReceivedRequest& request);
274
275     void impl_eventBuffer_flush(IrisReceivedRequest& request);
```

```
276
277     void impl_eventBuffer_destroy(IrisReceivedRequest& request);
278
279     void impl_ec_eventBuffer(IrisReceivedRequest& request);
280
281     void register_ec_IRIS_INSTANCE_REGISTRY_CHANGED();
282     IrisErrorCode ec_IRIS_INSTANCE_REGISTRY_CHANGED(EventStreamId esId, const IrisValueMap& fields,
        uint64_t time,
283                                                     InstanceId sInstId, bool syncEc, std::string&
        errorMessageOut);
284
286
287     void impl_event_registerProxyEventSource(IrisReceivedRequest& request);
288
289     void impl_event_unregisterProxyEventSource(IrisReceivedRequest& request);
290
291     void impl_eventStream_create_proxy(IrisReceivedRequest& request);
292
293     IrisErrorCode impl_eventStream_destroy_target(IrisReceivedRequest& request, EventStream* evStream);
294
295     void impl_eventStream_enable_proxy(IrisReceivedRequest& request, EventStream* evStream);
296
297     void impl_eventStream_disable_proxy(IrisReceivedRequest& request, EventStream* evStream);
298
299     void impl_eventStream_getCounter_proxy(IrisReceivedRequest& request, EventStream* evStream);
300
301     void impl_eventStream_setTraceRanges_proxy(IrisReceivedRequest& request, EventStream* evStream);
302
303     void impl_eventStream_getState_proxy(IrisReceivedRequest& request, EventStream* evStream);
304
305     void impl_eventStream_flush_proxy(IrisReceivedRequest& request, EventStream* evStream);
306
307     void impl_eventStream_setOptions_proxy(IrisReceivedRequest& request, EventStream* evStream);
308
309     void impl_eventStream_action_proxy(IrisReceivedRequest& request, EventStream* evStream);
310
311     ProxyEventInfo& getProxyEventInfo(EventStream* evStream);
312
313     InstanceId getTargetInstId(EventStream* evStream);
314
316
317     EventStream* getEventStream(EventStreamId esId);
318
319     struct EventBufferStreamInfo;
320     struct EventBuffer;
321
324     const EventBufferStreamInfo* getEventBufferStreamInfo(InstanceId sInstId, EventStreamId esId) const;
325
327     EventBuffer* getEventBuffer(EventBufferId evBufId) const;
328
331     void eventBufferSend(EventBuffer *eventBuffer, bool flush);
332
334     void eventBufferDestroy(EventBufferId evBufId);
335
336     //Find a free event stream ID where a new EventStream can be added
337     //The returned ID is greater than or equal to 'minEsId'
338     EventStreamId findFreeEventStreamId(EventStreamId minEsId);
339
341
343     IrisInstance* irisInstance;
344
346     std::vector<EventSourceInfoAndDelegate> eventSources;
347
349     std::map<std::string, uint64_t>         srcNameToId;
350
353     std::vector<EventStream*> eventStreams;
354
357     std::vector<EventStreamId> freeEsIds;
358
361     std::list<EventStreamId> inUseEsIds;
362
363 public:
364     bool hasEventStreams() const { return inUseEsIds.size() > 0; }
365
366 private:
367
369     EventStreamCreateDelegate defaultEsCreateDelegate;
370
372     IrisLogger log;
373
378     bool instance_registry_changed_registered{};
379
381     struct EventStreamOriginInfo
382     {
383         EventStreamId esId;
384         InstanceId sInstId;
385     };
```

```
386
388     struct EventBuffer
389     {
391         EventBuffer(const std::string& mode, uint64_t bufferSize, const std::string& ebcFunc, InstanceId
        ebcInstId, bool syncEbc, EventBufferId evBufId, IrisInstanceEvent *parent);
392
394         ~EventBuffer();
395
397         void clear();
398
400         const uint64_t* getResponse(RequestId requestId);
401
405         void getRequest(bool flush);
406
408         void addEventData(EventStreamInfoId esInfoId, uint64_t time, const uint64_t *fieldsU64Json);
409
411         void dropOldEvents(uint64_t targetBufferSizeU64);
412
414         std::string mode;
415
417         uint64_t bufferSizeU64{};
418
420         std::string ebcFunc;
421
423         InstanceId ebcInstId{IRIS_UINT64_MAX};
424
426         bool syncEbc{};
427
429         std::vector<EventStreamOriginInfo> eventStreams;
430
456         IrisU64JsonWriter writer;
457
459         uint64_t numEvents{};
460
462         size_t eventDataStartPos{};
463
466         IrisU64JsonWriter responseHeader;
467         size_t responseStartPos{};
468         size_t responseObjectPos{};
469         size_t responseArrayPos{};
470
473         IrisU64JsonWriter requestHeader;
474         size_t requestStartPos{};
475         size_t requestParamsPos{};
476         size_t requestReasonPos{};
477         size_t requestArrayPos{};
478
479         const uint64_t reasonSend  = 0x200000646E657304; // == "send"
480         const uint64_t reasonFlush = 0x20006873756C6605; // == "flush"
481
483         IrisInstanceEvent *parent{};
484     };
485     friend struct EventBuffer;
486
490     std::vector<EventBuffer*> eventBuffers;
491
494     std::vector<EventBufferId> freeEventBufferIds;
495
498     struct EventBufferStreamInfo
499     {
500         EventBuffer* eventBuffer{};
501         EventStreamInfoId esInfoId{};
502     };
503
511     std::vector<std::vector<EventBufferStreamInfo» eventCallbackInfoToEventBufferStreamInfo;
512
514     bool inEventStreamCreate{};
515 };
516
522 class EventStream
523 {
524 public:
528     EventStream()
529     {
530     }
531
532     virtual ~EventStream()
533     {
534         // Detach fieldObj from writer contained in internal_req so it does not touch
535         // internal_req after it was deleted.
536         //
537         // Background:
538         // IrisEventRegistry first calls emitEventBegin() on all event streams and one
539         // of the callbacks may lead to the destruction of the destination instance which
540         // will destroy all event streams, including the ones which had emitEventBegin()
541         // called on them without matching emitEventEnd().
542         // While such an event stream is deleted (with this destructor) fieldObj would try
```

```
543         // to make the field object consisent, after the writer was deleted. To prevent that,
544         // we detach fieldObj from the writer so fieldObj does nothing on destruction.
545         fieldObj.detach();
546 
547         delete internal_req;
548     }
549 
561     void selfRelease()
562     {
563         // Disable the event stream if it is still enabled.
564         if (isEnabled())
565         {
566             disable();
567         }
568 
569         // The request to destroy this event stream is nested and processed in the delegate to
570         // wait for the response, so it is not multi-threaded and no need to protect the variables.
571         if (!isInEventCallback)
572         {
573             delete this;
574             return;
575         }
576 
577         // We are currently in an event callback.
578         // Cancel the wait and release this object later when the callback returns.
579         req->cancel();
580         selfReleaseAfterReturnFromEventCallback = true;
581     }
582 
593     virtual IrisErrorCode enable() = 0;
594 
605     virtual IrisErrorCode disable() = 0;
606 
616     virtual IrisErrorCode getState(IrisValueMap& fields)
617     {
618         (void) fields;
619         return E_not_supported_for_event_source;
620     }
621 
631     virtual IrisErrorCode flush(RequestId requestId)
632     {
633         (void)requestId;
634         return E_not_supported_for_event_source;
635     }
636 
654     virtual IrisErrorCode setOptions(const AttributeValueMap& options, bool eventStreamCreate,
    std::string& errorMessageOut)
655     {
656         (void)options;
657         (void)eventStreamCreate;
658         (void)errorMessageOut;
659 
660         // Event streams which do not support options happily accept an empty options map.
661         return options.empty() ? E_ok : E_not_supported_for_event_source;
662     }
663 
674     virtual IrisErrorCode action(const BreakpointAction& action_)
675     {
676         (void)action_;
677         return E_not_supported_for_event_source;
678     }
679 
680     // Temporary: Keep PVModelLib happy. TODO: Remove.
681     virtual IrisErrorCode insertTrigger()
682     {
683         return E_not_supported_for_event_source;
684     }
685 
686 
687     // --- Functions for basic properties ---
688 
705     void setProperties(IrisInstance* irisInstance, IrisInstanceEvent* irisInstanceEvent, EventSourceId
    evSrcId,
706                        InstanceId ecInstId, const std::string& ecFunc, EventStreamId esId,
707                        bool syncEc);
708 
714     bool isEnabled() const
715     {
716         return enabled;
717     }
718 
724     EventStreamId getEsId() const
725     {
726         return esId;
727     }
728 
734     const EventSourceInfo* getEventSourceInfo() const
```

```
735     {
736         return irisInstanceEvent ? irisInstanceEvent->getEventSourceInfo(evSrcId) : nullptr;
737     }
738
744     EventSourceId getEventSourceId() const { return evSrcId; }
745
752     InstanceId getEcInstId() const
753     {
754         return ecInstId;
755     }
756
757     // --- Functions for the counter mode ---
758
765     void setCounter(uint64_t startVal, const EventCounterMode& counterMode);
766
772     bool isCounter() const
773     {
774         return counter;
775     }
776
781     void setProxyForOtherInstance()
782     {
783         isProxyForOtherInstance = true;
784     }
785
791     bool IsProxyForOtherInstance() const
792     {
793         return isProxyForOtherInstance;
794     }
795
801     void setProxiedByInstanceId(InstanceId instId)
802     {
803         proxiedByInstanceId = instId;
804     }
805
811     bool IsProxiedByOtherInstance() const
812     {
813         return proxiedByInstanceId != IRIS_UINT64_MAX;
814     }
815
821     InstanceId getProxiedByInstanceId() const
822     {
823         return proxiedByInstanceId;
824     }
825
831     uint64_t getCountVal() const
832     {
833         return curVal;
834     }
835
836     // --- Functions for event stream with ranges
837
846     IrisErrorCode setRanges(const std::string& aspect, const std::vector<uint64_t>& ranges);
847
856     bool checkRangePc(uint64_t pc) const
857     {
858         return ranges.empty() || (aspect != ":pc") || checkRangesHelper(pc, ranges);
859     }
860
861     // --- Functions to emit the event callback ---
862     // Usage (example):
863     //     emitEventBegin(time, pc);    // Start to emit the callback.
864     //     addField(...);           // Add field value.
865     //     addField(...);           // Add field value.
866     //     ...
867     //     emitEventEnd();              // Emit the callback.
868
876     void emitEventBegin(IrisRequest& req, uint64_t time, uint64_t pc = IRIS_UINT64_MAX);
877
884     void emitEventBegin(uint64_t time, uint64_t pc = IRIS_UINT64_MAX);
885
895     void addField(const IrisU64StringConstant& field, uint64_t value)
896     {
897         addFieldRangeHelper(field, value);
898     }
899
909     void addField(const IrisU64StringConstant& field, int64_t value)
910     {
911         addFieldRangeHelper(field, value);
912     }
913
923     void addField(const IrisU64StringConstant& field, bool value)
924     {
925         addFieldRangeHelper(field, value);
926     }
927
937     template <class T>
```

```
938     void addField(const IrisU64StringConstant& field, const T& value)
939     {
940         fieldObj.member(field, value);
941     }
942
952     void addField(const IrisU64StringConstant& field, const uint8_t *data, size_t sizeInBytes)
953     {
954         fieldObj.member(field, data, sizeInBytes);
955     }
956
966     void addFieldSlow(const std::string& field, uint64_t value)
967     {
968         addFieldSlowRangeHelper(field, value);
969     }
970
980     void addFieldSlow(const std::string& field, int64_t value)
981     {
982         addFieldSlowRangeHelper(field, value);
983     }
984
994     void addFieldSlow(const std::string& field, bool value)
995     {
996         addFieldSlowRangeHelper(field, value);
997     }
998
1008     template <class T>
1009     void addFieldSlow(const std::string& field, const T& value)
1010     {
1011         fieldObj.memberSlow(field, value);
1012     }
1013
1023     void addFieldSlow(const std::string& field, const uint8_t *data, size_t sizeInBytes)
1024     {
1025         fieldObj.memberSlow(field, data, sizeInBytes);
1026     }
1027
1037     void emitEventEnd(bool send = true);
1038
1039 private:
1041
1045     bool counterTrigger();
1046
1048     bool checkRanges() const
1049     {
1050         return !aspectFound || checkRangesHelper(curAspectValue, ranges);
1051     }
1052
1054     static bool checkRangesHelper(uint64_t value, const std::vector<uint64_t>& ranges);
1055
1057     template <typename T>
1058     void addFieldRangeHelper(const IrisU64StringConstant& field, T value)
1059     {
1060         if (!aspect.empty() && aspect == toString(field))
1061         {
1062             aspectFound    = true;
1063             curAspectValue = static_cast<uint64_t>(value);
1064         }
1065
1066         fieldObj.member(field, value);
1067     }
1068
1070     template <typename T>
1071     void addFieldSlowRangeHelper(const std::string& field, T value)
1072     {
1073         if (aspect == field)
1074         {
1075             aspectFound    = true;
1076             curAspectValue = static_cast<uint64_t>(value);
1077         }
1078
1079         fieldObj.memberSlow(field, value);
1080     }
1081
1082 protected:
1084
1086     IrisInstance* irisInstance{};
1087
1089     IrisInstanceEvent* irisInstanceEvent{};
1090
1092     EventSourceId evSrcId{IRIS_UINT64_MAX};
1093
1095     InstanceId ecInstId{IRIS_UINT64_MAX};
1096
1098     std::string ecFunc;
1099
1101     EventStreamId esId{IRIS_UINT64_MAX};
1102
```

```
1104     bool syncEc{};
1105
1107     bool enabled{};
1108
1110     IrisRequest*              req{};
1111     IrisRequest*              internal_req{};
1112     IrisU64JsonWriter::Object fieldObj;
1113
1115
1117     bool counter{};
1118
1120     uint64_t startVal{};
1121     uint64_t curVal{};
1122
1124     EventCounterMode counterMode{};
1125
1127
1128     std::string           aspect;
1129     std::vector<uint64_t> ranges;
1130
1132     bool aspectFound{};
1133
1135     uint64_t curAspectValue{};
1136
1138     bool isProxyForOtherInstance{false};
1139
1142     InstanceId proxiedByInstanceId{IRIS_UINT64_MAX};
1143
1144 private:
1146     int isInEventCallback{};
1147
1149     bool selfReleaseAfterReturnFromEventCallback{};
1150 };
1151
1155 class IrisEventStream : public EventStream
1156 {
1157 public:
1158     IrisEventStream(IrisEventRegistry* registry_);
1159
1160     virtual IrisErrorCode enable() override;
1161
1162     virtual IrisErrorCode disable() override;
1163
1164 private:
1165     IrisEventRegistry* registry;
1166 };
1167
1171 class IrisEventRegistry
1172 {
1173 public:
1179     bool empty() const
1180     {
1181         return esSet.empty();
1182     }
1183
1190     bool registerEventStream(EventStream* evStream);
1191
1198     bool unregisterEventStream(EventStream* evStream);
1199
1200     // --- Functions to emit the callback of all registered event streams ---
1201     // Usage (example):
1202     //     emitEventBegin(time, pc);    // Start to emit the callback.
1203     //     addField(...);          // Add field value.
1204     //     addField(...);          // Add field value.
1205     //     ...
1206     //     emitEventEnd();             // Emit the callback.
1207
1208     void emitEventBegin(uint64_t time, uint64_t pc = IRIS_UINT64_MAX) const;
1209
1220     template <class T>
1221     void addField(const IrisU64StringConstant& field, const T& value) const
1222     {
1223         for (std::set<EventStream*>::const_iterator i = esSet.begin(), e = esSet.end(); i != e; i++)
1224             (*i)->addField(field, value);
1225     }
1226
1237     template <class T>
1238     void addFieldSlow(const std::string& field, const T& value) const
1239     {
1240         for (std::set<EventStream*>::const_iterator i = esSet.begin(), e = esSet.end(); i != e; i++)
1241             (*i)->addFieldSlow(field, value);
1242     }
1243
1268     template <class T, typename F>
1269     void forEach(F && func) const
1270     {
1271         for (std::set<EventStream*>::const_iterator i = esSet.begin(), e = esSet.end(); i != e; i++)
```

```
1272         {
1273             T* t = static_cast<T*>(*i);
1274             func(*t);
1275         }
1276     }
1277
1283     void emitEventEnd() const;
1284
1285     typedef std::set<EventStream*>::const_iterator iterator;
1286
1294     iterator begin() const
1295     {
1296         return esSet.begin();
1297     }
1298
1306     iterator end() const
1307     {
1308         return esSet.end();
1309     }
1310
1311     ~IrisEventRegistry()
1312     {
1313         // Disable any remaining event streams.
1314         // Calling disable() on an EventStream will cause esSet to be modified so we need to loop
       without
1315         // using iterators which become invalidated.
1316         while (!esSet.empty())
1317         {
1318             (*esSet.begin())->disable();
1319         }
1320     }
1321
1322 private:
1323     // All registered event streams
1324     std::set<EventStream*> esSet;
1325 };
1326
1327 NAMESPACE_IRIS_END
1328
1329 #endif // #ifndef ARM_INCLUDE_IrisInstanceBreakpoint_h
```

9.29 IrisInstanceFactoryBuilder.h File Reference

A helper class to build instantiation parameter metadata.

```
#include "iris/IrisParameterBuilder.h"
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisObjects.h"
#include <string>
#include <vector>
```

Classes

- class iris::IrisInstanceFactoryBuilder

  A builder class to construct instantiation parameter metadata.

9.29.1 Detailed Description

A helper class to build instantiation parameter metadata.

Copyright

9.30 IrisInstanceFactoryBuilder.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisInstanceFactoryBuilder_h
8 #define ARM_INCLUDE_IrisInstanceFactoryBuilder_h
9
10 #include "iris/IrisParameterBuilder.h"
```

```
11 #include "iris/detail/IrisCommon.h"
12 #include "iris/detail/IrisObjects.h"
13
14 #include <string>
15 #include <vector>
16
17 NAMESPACE_IRIS_START
18
22 class IrisInstanceFactoryBuilder
23 {
24 private:
26     std::vector<ResourceInfo> parameters;
27
29     std::vector<ResourceInfo> hidden_parameters;
30
32     std::string parameter_prefix;
33
34     ResourceInfo& addParameterInternal(const std::string& name, uint64_t bitWidth, const std::string&
      description,
35                                        const std::string& type, bool hidden)
36     {
37         std::vector<ResourceInfo>& param_list = hidden ? hidden_parameters : parameters;
38         param_list.resize(parameters.size() + 1);
39         ResourceInfo& info = param_list.back();
40
41         info.name     = name;
42         info.bitWidth = bitWidth;
43         info.description    = description;
44         info.type     = type;
45
46         return info;
47     }
48
49 public:
55     IrisInstanceFactoryBuilder(const std::string& prefix)
56         : parameter_prefix(prefix)
57     {
58     }
59
68     IrisParameterBuilder addParameter(const std::string& name, uint64_t bitWidth, const std::string&
      description)
69     {
70         return IrisParameterBuilder(addParameterInternal(parameter_prefix + name, bitWidth, description,
      "" /*numeric*/, false));
71     }
72
81     IrisParameterBuilder addHiddenParameter(const std::string& name, uint64_t bitWidth, const
      std::string& description)
82     {
83         return IrisParameterBuilder(addParameterInternal(parameter_prefix + name, bitWidth, description,
      "" /*numeric*/, true));
84     }
85
93     IrisParameterBuilder addStringParameter(const std::string& name, const std::string& description)
94     {
95         return IrisParameterBuilder(addParameterInternal(parameter_prefix + name, 0, description,
      "string", false));
96     }
97
105     IrisParameterBuilder addHiddenStringParameter(const std::string& name, const std::string&
      description)
106     {
107         return IrisParameterBuilder(addParameterInternal(parameter_prefix + name, 0, description,
      "string", true));
108     }
109
120     IrisParameterBuilder addBoolParameter(const std::string& name, const std::string& description)
121     {
122         ResourceInfo& info = addParameterInternal(parameter_prefix + name, 1, description, "numeric",
      false);
123
124         // Be explicit about the range even though there are only two possible values anyway.
125         info.parameterInfo.min.push_back(0);
126         info.parameterInfo.max.push_back(1);
127
128         // Add enum strings for the values
129         info.enums.push_back(EnumElementInfo(IrisValue(0), "false", ""));
130         info.enums.push_back(EnumElementInfo(IrisValue(1), "true", ""));
131
132         return IrisParameterBuilder(info);
133     }
134     [[deprecated("Use addBoolParameter() instead.")]] IrisParameterBuilder addBooleanParameter(const
      std::string& name, const std::string& description)
135     {
136         return addBoolParameter(name, description);
137     }
138
```

```
149     IrisParameterBuilder addHiddenBoolParameter(const std::string& name, const std::string& description)
150     {
151         ResourceInfo& info = addParameterInternal(parameter_prefix + name, 1, description, "numeric",
        true);
152 
153         // Be explicit about the range even though there are only two possible values anyway.
154         info.parameterInfo.min.push_back(0);
155         info.parameterInfo.max.push_back(1);
156 
157         // Add enum strings for the values
158         info.enums.push_back(EnumElementInfo(IrisValue(0), "false", ""));
159         info.enums.push_back(EnumElementInfo(IrisValue(1), "true", ""));
160 
161         return IrisParameterBuilder(info);
162     }
163     [[deprecated("Use addHiddenBoolParameter() instead.")]] IrisParameterBuilder
       addHiddenBooleanParameter(const std::string& name, const std::string& description)
164     {
165         return addHiddenBoolParameter(name, description);
166     }
167 
174     const std::vector<ResourceInfo>& getParameterInfo() const
175     {
176         return parameters;
177     }
178 
185     const std::vector<ResourceInfo>& getHiddenParameterInfo() const
186     {
187         return hidden_parameters;
188     }
189 };
190 
191 NAMESPACE_IRIS_END
192 
193 #endif // ARM_INCLUDE_IrisInstanceFactoryBuilder_h
```

9.31 IrisInstanceImage.h File Reference

Image-loading add-on to IrisInstance and image-loading callback add-on to the caller.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include <cstdio>
```

Classes

- class iris::IrisInstanceImage

  Image loading add-on for IrisInstance.
- class iris::IrisInstanceImage_Callback

  Image loading add-on for IrisInstance clients implementing image_loadDataRead().

Typedefs

- typedef IrisDelegate< const std::vector< uint8_t > & > iris::ImageLoadDataDelegate

  Delegate to load an image from the given data.
- typedef IrisDelegate< const std::string & > iris::ImageLoadFileDelegate

  Delegate function to load an image from the given file.

9.31.1 Detailed Description

Image-loading add-on to IrisInstance and image-loading callback add-on to the caller.

The IrisInstanceImage class:

- Implements all image-loading Iris functions.
- Maintains and provides image metadata, for example path, instanceSideFile, rawAddr.
- Converts between Iris image-loading functions (image_load∗()) and various C++ access functions.

9.31.2 Typedef Documentation

9.31.2.1 ImageLoadDataDelegate

```
typedef IrisDelegate<const std::vector<uint8_t>&> iris::ImageLoadDataDelegate
```

Delegate to load an image from the given data.

```
IrisErrorCode loadImage(const std::vector<uint8_t> &data)
```

Typical implementations try to load the data with the supported formats.
Errors:

- If the image format is unknown, E_unknown_image_format is returned.
- If the image format is known but the image could not be loaded, E_image_format_error is returned.

9.31.2.2 ImageLoadFileDelegate

```
typedef IrisDelegate<const std::string&> iris::ImageLoadFileDelegate
```

Delegate function to load an image from the given file.
The path can be absolute or relative to the current working directory.

```
IrisErrorCode loadImage(const std::string &path)
```

Typical implementations try to load the file with the supported formats.
Errors:

- If the file specified by path could not be opened, E_error_opening_file is returned.
- If the file could be opened but could not be read, E_io_error is returned.
- If the image format is unknown, E_unknown_image_format is returned.
- If the image format is known but the image could not be loaded, E_image_format_error is returned.

9.32 IrisInstanceImage.h

Go to the documentation of this file.

```
1
13 #ifndef ARM_INCLUDE_IrisInstanceImage_h
14 #define ARM_INCLUDE_IrisInstanceImage_h
15
16 #include "iris/detail/IrisCommon.h"
17 #include "iris/detail/IrisDelegate.h"
18 #include "iris/detail/IrisLogger.h"
19 #include "iris/detail/IrisObjects.h"
20
21 #include <cstdio>
22
23 NAMESPACE_IRIS_START
24
25 class IrisInstance;
26 class IrisReceivedRequest;
27
44 typedef IrisDelegate<const std::string&> ImageLoadFileDelegate;
45
59 typedef IrisDelegate<const std::vector<uint8_t>&> ImageLoadDataDelegate;
60
```

```
77 class IrisInstanceImage
78 {
79
80 public:
86     IrisInstanceImage(IrisInstance* irisInstance = 0);
87
93     void attachTo(IrisInstance* irisInstance);
94
100     void setLoadImageFileDelegate(ImageLoadFileDelegate delegate);
101
107     void setLoadImageDataDelegate(ImageLoadDataDelegate delegate);
108
116     static IrisErrorCode readFileData(const std::string& fileName, std::vector<uint8_t>& data);
117
118 private:
120     void loadImageFromData(IrisReceivedRequest& request, const ImageReadResult& imageData);
121
123
125     void impl_image_loadFile(IrisReceivedRequest& request);
126
128     void impl_image_loadData(IrisReceivedRequest& request);
129
131     void impl_image_loadDataPull(IrisReceivedRequest& request);
132
133     void impl_image_getMetaInfoList(IrisReceivedRequest& request);
134
135     void impl_image_clearMetaInfoList(IrisReceivedRequest& request);
136
138     void writeRawDataToMemory(IrisReceivedRequest& request, const std::vector<uint8_t>& data, uint64_t
    rawAddr, MemorySpaceId rawSpaceId);
139
141     IrisErrorCode pullData(InstanceId callerId, uint64_t tag, ImageReadResult& result);
142
144
146     IrisInstance* irisInstance;
147
151     typedef std::vector<ImageMetaInfo> ImageMetaInfoList;
152     ImageMetaInfoList                  metaInfos;
153
155     IrisLogger log;
156
157     ImageLoadFileDelegate loadFileDelegate;
158     ImageLoadDataDelegate loadDataDelegate;
159 };
160
171 class IrisInstanceImage_Callback
172 {
173 public:
179     IrisInstanceImage_Callback(IrisInstance* irisInstance = 0);
180
181     ~IrisInstanceImage_Callback();
182
188     void attachTo(IrisInstance* irisInstance);
189
199     uint64_t openImage(const std::string& fileName);
200
201 protected:
203     void impl_image_loadDataRead(IrisReceivedRequest& request);
204
205 private:
207     IrisErrorCode readImageData(uint64_t tag, uint64_t position, uint64_t size, bool end,
    ImageReadResult& result);
208
210     IrisInstance* irisInstance;
211
213     IrisLogger log;
214
216     typedef std::vector<FILE*> ImageList;
217     ImageList                  images;
218 };
219
220 NAMESPACE_IRIS_END
221
222 #endif // #ifndef ARM_INCLUDE_IrisInstanceImage_h
```

9.33 IrisInstanceMemory.h File Reference

Memory add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
```

Classes

- struct iris::IrisInstanceMemory::AddressTranslationInfoAndAccess

  Contains static address translation information.
- class iris::IrisInstanceMemory

  Memory add-on for IrisInstance.
- struct iris::IrisInstanceMemory::SpaceInfoAndAccess

  Entry in 'spaceInfos'.

Typedefs

- typedef IrisDelegate< uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult & > iris::MemoryAddressTranslateDelegat

  Delegate to translate an address.
- typedef IrisDelegate< const MemorySpaceInfo &, uint64_t, const IrisValueMap &, const std::vector< std::string > &, IrisValueMap & > iris::MemoryGetSidebandInfoDelegate
- typedef IrisDelegate< const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, MemoryReadResult & > iris::MemoryReadDelegate

  Delegate to read memory data.
- typedef IrisDelegate< const MemorySpaceInfo &, uint64_t, uint64_t, uint64_t, const AttributeValueMap &, const uint64_t *, MemoryWriteResult & > iris::MemoryWriteDelegate

  Delegate to write memory data.

9.33.1 Detailed Description

Memory add-on to IrisInstance.

Copyright

Copyright (C) 2015 Arm Limited. All rights reserved.

The IrisInstanceMemory class:

- Implements all memory-related Iris functions.
- Feeds memory-related properties (memory.*) to instance_getProperties() of the associated IrisInstance.
- Provides infrastructure that is useful for Iris clients.
- Maintains and provides memory meta information (memory spaces, address translations, sideband information).
- Converts between Iris memory access functions (memory_read()) and various C++ access functions.

9.33.2 Typedef Documentation

9.33.2.1 MemoryAddressTranslateDelegate

```
typedef IrisDelegate<uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult&> iris::MemoryAddressTransl
```

Delegate to translate an address.

```
IrisErrorCode translate(MemorySpaceId inSpaceId, uint64_t address,
                        MemorySpaceId outSpaceId, MemoryAddressTranslationResult &result)
```

inSpaceId, address, and outSpaceId are guaranteed to be valid.
Typical implementations inspect the inSpaceId and outSpaceId to determine how to translate the address.
Return addresses are appended to result.address, which is a vector<uint64_t>:

- If this array is empty then 'address' is not mapped in 'outSpaceId'.

- If the array contains exactly one element then the mapping is unique.
- If it contains multiple addresses then 'address' is accessible in the same way under all of these addresses in 'outSpaceId'.

Error: Return E_* error code for translation errors.

9.33.2.2 MemoryGetSidebandInfoDelegate

```
typedef IrisDelegate<const MemorySpaceInfo&, uint64_t, const IrisValueMap&, const std::vector<std↩
::string>&, IrisValueMap&> iris::MemoryGetSidebandInfoDelegate
```

@ Delegate to get memory sideband information.

```
IrisErrorCode getSidebandInfo(const MemorySpaceInfo &spaceInfo, uint64_t address,
                              const IrisValueMap &attrib,
                              const std::vector<std::string> &request,
                              IrisValueMap &result)
```

Returns sideband information for a range of addresses in a given memory space.

9.33.2.3 MemoryReadDelegate

```
typedef IrisDelegate<const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t, const Attribute↩
ValueMap&, MemoryReadResult&> iris::MemoryReadDelegate
```

Delegate to read memory data.

```
IrisErrorCode read(const MemorySpaceInfo &spaceInfo, uint64_t address, uint64_t byteWidth,
                   uint64_t count, const AttributeValueMap &attrib, MemoryReadResult &result)
```

spaceInfo, address, byteWidth, and count are guaranteed to be valid.
Typical implementations inspect the spaceId, address, byteWidth, and count to determine which memory elements should be read. Then they append the read elements to result.data, which is a vector<uint64_t>:

- Data elements are read from ascending addresses, packed into uint64_ts such that the lowest address is in the lowest bits.
- Elements of byteWidth >= 2 are read with the endianness of the memory space inside each element, but elements are stored with the lowest bits inside each uint64_t (for byteWidth < 8) and with the lowest bits first in sequences of uint64_t (for byteWidth > 8).

Error: Return E_* error code for read errors. It appends the address that could not be read to result.error.

9.33.2.4 MemoryWriteDelegate

```
typedef IrisDelegate<const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t, const Attribute↩
ValueMap&, const uint64_t*, MemoryWriteResult&> iris::MemoryWriteDelegate
```

Delegate to write memory data.

```
IrisErrorCode write(const MemorySpaceInfo &spaceInfo, uint64_t address, uint64_t byteWidth,
                    uint64_t count, const AttributeValueMap &attrib, const uint64_t *data, MemoryWriteResult
       &result)
```

See also

MemoryReadDelegate data contains the data elements to be written in the same format as MemoryRead↩ Result.data for reads.

9.34 IrisInstanceMemory.h

Go to the documentation of this file.

```
1
14 #ifndef ARM_INCLUDE_IrisInstanceMemory_h
15 #define ARM_INCLUDE_IrisInstanceMemory_h
16
17 #include "iris/detail/IrisCommon.h"
18 #include "iris/detail/IrisDelegate.h"
19 #include "iris/detail/IrisLogger.h"
20 #include "iris/detail/IrisObjects.h"
21
22 NAMESPACE_IRIS_START
23
24 class IrisInstance;
25 class IrisReceivedRequest;
26
```

```
47 typedef IrisDelegate<const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
48                       const AttributeValueMap&, MemoryReadResult&>
49     MemoryReadDelegate;
50
61 typedef IrisDelegate<const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
62                       const AttributeValueMap&, const uint64_t*, MemoryWriteResult&>
63     MemoryWriteDelegate;
64
85 typedef IrisDelegate<uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult&>
       MemoryAddressTranslateDelegate;
86
99 typedef IrisDelegate<const MemorySpaceInfo&, uint64_t, const IrisValueMap&,
100                       const std::vector<std::string>&, IrisValueMap&>
101     MemoryGetSidebandInfoDelegate;
102
129 class IrisInstanceMemory
130 {
131 public:
137     struct SpaceInfoAndAccess
138     {
139         MemorySpaceInfo               spaceInfo;
140         MemoryReadDelegate            readDelegate;     // May be empty. In this case
       defaultReadDelegate is used.
141         MemoryWriteDelegate           writeDelegate;    // May be empty. In this case
       defaultWriteDelegate is used.
142         MemoryGetSidebandInfoDelegate sidebandDelegate; // May be empty. In this case sidebandDelegate
       is used.
143     };
144
148     struct AddressTranslationInfoAndAccess
149     {
150         AddressTranslationInfoAndAccess(MemorySpaceId inSpaceId, MemorySpaceId outSpaceId, const
       std::string& description)
151             : translationInfo(inSpaceId, outSpaceId, description)
152         {
153         }
154
155         MemorySupportedAddressTranslationResult translationInfo;
156         MemoryAddressTranslateDelegate          translateDelegate;
157     };
158
165     IrisInstanceMemory(IrisInstance* irisInstance = 0);
166
172     void attachTo(IrisInstance* irisInstance);
173
179     void setDefaultReadDelegate(MemoryReadDelegate delegate = MemoryReadDelegate())
180     {
181         memReadDelegate = delegate;
182     }
183
189     void setDefaultWriteDelegate(MemoryWriteDelegate delegate = MemoryWriteDelegate())
190     {
191         memWriteDelegate = delegate;
192     }
193
201     SpaceInfoAndAccess& addMemorySpace(const std::string& name);
202
213     AddressTranslationInfoAndAccess& addAddressTranslation(MemorySpaceId inSpaceId, MemorySpaceId
       outSpaceId,
214                                                            const std::string& description);
215
221     void setDefaultTranslateDelegate(MemoryAddressTranslateDelegate delegate =
       MemoryAddressTranslateDelegate())
222     {
223         translateDelegate = delegate;
224     }
225
231     void setDefaultGetSidebandInfoDelegate(MemoryGetSidebandInfoDelegate delegate =
       MemoryGetSidebandInfoDelegate())
232     {
233         if (delegate.empty())
234         {
235             delegate = MemoryGetSidebandInfoDelegate::make<IrisInstanceMemory,
       &IrisInstanceMemory::getDefaultSidebandInfo>(this);
236         }
237
238         sidebandDelegate = delegate;
239     }
240
241 private:
243
244     void impl_memory_getMemorySpaces(IrisReceivedRequest& request);
245
246     void impl_memory_read(IrisReceivedRequest& request);
247
248     void impl_memory_write(IrisReceivedRequest& request);
249
```

```
250     void impl_memory_translateAddress(IrisReceivedRequest& request);
251
252     void impl_memory_getUsefulAddressTranslations(IrisReceivedRequest& request);
253
254     void impl_memory_getSidebandInfo(IrisReceivedRequest& request);
256
258     IrisErrorCode getDefaultSidebandInfo(const MemorySpaceInfo& spaceInfo, uint64_t address,
259                                          const IrisValueMap&             attrib,
260                                          const std::vector<std::string>& request,
261                                          IrisValueMap&                   result);
262
264     bool checkAddress(IrisReceivedRequest& request, uint64_t address, const MemorySpaceInfo& spaceInfo);
265
266     // --- state ---
267
269     IrisInstance* irisInstance;
270
272     typedef std::vector<SpaceInfoAndAccess> SpaceInfoList;
273     SpaceInfoList                           spaceInfos;
274
276     typedef std::vector<AddressTranslationInfoAndAccess> SupportedTranslations;
277     SupportedTranslations                                supportedTranslations;
278
280     MemoryReadDelegate             memReadDelegate;
281     MemoryWriteDelegate            memWriteDelegate;
282     MemoryAddressTranslateDelegate translateDelegate;
283
286     MemoryGetSidebandInfoDelegate sidebandDelegate;
287
289     IrisLogger log;
290 };
291
292 NAMESPACE_IRIS_END
293
294 #endif // #ifndef ARM_INCLUDE_IrisInstanceMemory_h
```

9.35 IrisInstancePerInstanceExecution.h File Reference

Per-instance execution control add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include <cstdio>
```

Classes

- class iris::IrisInstancePerInstanceExecution

  Per-instance execution control add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< bool & > iris::PerInstanceExecutionStateGetDelegate

  Get the execution state.
- typedef IrisDelegate< bool > iris::PerInstanceExecutionStateSetDelegate

  Delegate to set the execution state.

9.35.1 Detailed Description

Per-instance execution control add-on to IrisInstance.

Copyright

Implements all per-instance execution control-related Iris functions.

9.35.2 Typedef Documentation

9.35.2.1 PerInstanceExecutionStateGetDelegate

```
typedef IrisDelegate<bool&> iris::PerInstanceExecutionStateGetDelegate
```

Get the execution state.
enabled should be set to true if execution is enabled and false otherwise.

```
IrisErrorCode getState(bool &enabled)
```

Return E_ok on success, otherwise return the error code.

9.35.2.2 PerInstanceExecutionStateSetDelegate

```
typedef IrisDelegate<bool> iris::PerInstanceExecutionStateSetDelegate
```

Delegate to set the execution state.
Enable or disable the execution of instructions (or processing of work items).

```
IrisErrorCode setState(bool enable)
```

Return E_ok on success, otherwise return the error code.

9.36 IrisInstancePerInstanceExecution.h

Go to the documentation of this file.

```
1
9  #ifndef ARM_INCLUDE_IrisInstancePerInstanceExecution_h
10 #define ARM_INCLUDE_IrisInstancePerInstanceExecution_h
11
12 #include "iris/detail/IrisCommon.h"
13 #include "iris/detail/IrisDelegate.h"
14 #include "iris/detail/IrisLogger.h"
15 #include "iris/detail/IrisObjects.h"
16
17 #include <cstdio>
18
19 NAMESPACE_IRIS_START
20
21 class IrisInstance;
22 class IrisReceivedRequest;
23
33 typedef IrisDelegate<bool> PerInstanceExecutionStateSetDelegate;
34
44 typedef IrisDelegate<bool&> PerInstanceExecutionStateGetDelegate;
45
53 class IrisInstancePerInstanceExecution
54 {
55 public:
61     IrisInstancePerInstanceExecution(IrisInstance* irisInstance = nullptr);
62
70     void attachTo(IrisInstance* irisInstance);
71
78     void setExecutionStateSetDelegate(PerInstanceExecutionStateSetDelegate delegate);
79
86     void setExecutionStateGetDelegate(PerInstanceExecutionStateGetDelegate delegate);
87
88 private:
89     void impl_perInstanceExecution_setState(IrisReceivedRequest& request);
90
91     void impl_perInstanceExecution_getState(IrisReceivedRequest& request);
92
94
96     IrisInstance* irisInstance;
97
99     PerInstanceExecutionStateSetDelegate execStateSet;
100     PerInstanceExecutionStateGetDelegate execStateGet;
101
103     IrisLogger log;
104 };
105
106 NAMESPACE_IRIS_END
107
108 #endif // #ifndef ARM_INCLUDE_IrisInstancePerInstanceExecution_h
```

9.37 IrisInstanceResource.h File Reference

Resource add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include <cassert>
```

Classes

- class iris::IrisInstanceResource

  Resource add-on for IrisInstance.
- struct iris::IrisInstanceResource::ResourceInfoAndAccess

  Entry in 'resourceInfos'.
- struct iris::ResourceWriteValue

Typedefs

- typedef IrisDelegate< const ResourceInfo &, ResourceReadResult & > iris::ResourceReadDelegate

  Delegate to read resources.
- typedef IrisDelegate< const ResourceInfo &, const ResourceWriteValue & > iris::ResourceWriteDelegate

  Delegate to write resources.

Functions

- uint64_t iris::resourceReadBitField (uint64_t parentValue, const ResourceInfo &resourceInfo)
- template<class T >
  void iris::resourceWriteBitField (T &parentValue, uint64_t fieldValue, const ResourceInfo &resourceInfo)

9.37.1 Detailed Description

Resource add-on to IrisInstance.

Copyright

The IrisInstanceResource class:

- Implements all resource-related Iris functions.
- Feeds resource-related properties (resource.*) to instance_getProperties() of the associated IrisInstance.
- Provides infrastructure that is useful for Iris clients.
- Maintains and provides resource meta information (name, bitwidth).
- Converts between Iris resource-access functions (resource_read()) and various C++ access functions.

9.37.2 Typedef Documentation

9.37.2.1 ResourceReadDelegate

```
typedef IrisDelegate<const ResourceInfo&, ResourceReadResult&> iris::ResourceReadDelegate
```

Delegate to read resources.

```
IrisErrorCode read(const ResourceInfo &resourceInfo, ResourceReadResult &result)
```

resourceInfo.rscId is guaranteed to be valid.
Typical implementations inspect the rscId, canonicalRn, addressOffset, or even the name or cname value to determine which resource should be read and then append the read data to result:

- Return data (no undefined bits):
  - Append data to result.data, which is a vector<uint64_t>. Append one uint64_t if resource is <= 64 bits.
  - Append multiple uint64_t for wider resources, least significant uint64_t first.
- Return data with undefined bits:
  - Same as above, but in addition, append a mask which contains 1 bit for all undefined bits to result.↩ undefinedBits (same format and length as result.data) and set all undefined bits to 0 in result.data.

Error: If the resource could not be read, return E_* error code, for example E_error_reading_write_only_resource, E_error_reading_resource, or E_not_implemented, and leave result unchanged.

9.37.2.2 ResourceWriteDelegate

```
typedef IrisDelegate<const ResourceInfo&, const ResourceWriteValue&> iris::ResourceWriteDelegate
```

Delegate to write resources.

```
IrisErrorCode write(const ResourceInfo &resourceInfo, const ResourceWriteValue &value)
```

resourceInfo.rscId is guaranteed to be valid.
Typical implementations inspect the rscId, canonicalRn, addressOffset, or even the name or cname value to determine which resource should be written.
`data` contains the data for all resources to be written in the same format as ResourceReadResult.data for reads. The number of elements in the data array is resourceInfo.getDataSizeInU64Chunks(). data is only evaluated for string resources.

9.37.3 Function Documentation

9.37.3.1 resourceReadBitField()

```
uint64_t iris::resourceReadBitField (
            uint64_t parentValue,
            const ResourceInfo & resourceInfo )  [inline]
```

Helper for ResourceReadDelegates to read a bit field of a parent register according to the lsbOffset and bitWidth in resourceInfo. This helps reducing redundancy in the debug interface implementation.

9.37.3.2 resourceWriteBitField()

```
template<class T >
void iris::resourceWriteBitField (
            T & parentValue,
            uint64_t fieldValue,
            const ResourceInfo & resourceInfo )  [inline]
```

Helper for ResourceWriteDelegates to write a bit field of a parent register according to the lsbOffset and bitWidth in resourceInfo. This helps reducing redundancy in the debug interface implementation.

9.38 IrisInstanceResource.h

Go to the documentation of this file.

```
1
14 #ifndef ARM_INCLUDE_IrisInstanceResource_h
15 #define ARM_INCLUDE_IrisInstanceResource_h
16
17 #include "iris/detail/IrisCommon.h"
18 #include "iris/detail/IrisDelegate.h"
19 #include "iris/detail/IrisLogger.h"
20 #include "iris/detail/IrisObjects.h"
21
22 #include <cassert>
23
24 NAMESPACE_IRIS_START
25
26 class IrisInstance;
27 class IrisReceivedRequest;
28
32 inline uint64_t resourceReadBitField(uint64_t parentValue, const ResourceInfo& resourceInfo)
33 {
34     return (resourceInfo.registerInfo.lsbOffset < 64) ?
35         ((parentValue » resourceInfo.registerInfo.lsbOffset) & maskWidthLsb(resourceInfo.bitWidth, 0))
36         : 0;
37 }
38
39
43 template<class T>
44 inline void resourceWriteBitField(T& parentValue, uint64_t fieldValue, const ResourceInfo& resourceInfo)
45 {
46     T mask = T(maskWidthLsb(resourceInfo.bitWidth, resourceInfo.registerInfo.lsbOffset));
47     parentValue &= ~mask;
48     parentValue |= T((resourceInfo.registerInfo.lsbOffset < 64) ?
49         ((fieldValue « resourceInfo.registerInfo.lsbOffset) & mask)
50         : 0);
51 }
52
53
58 struct ResourceWriteValue
59 {
60     const uint64_t*    data{};
61     const std::string* str{};
63 };
64
65
89 typedef IrisDelegate<const ResourceInfo&, ResourceReadResult&> ResourceReadDelegate;
90
106 typedef IrisDelegate<const ResourceInfo&, const ResourceWriteValue&> ResourceWriteDelegate;
107
120 class IrisInstanceResource
121 {
122 public:
128     struct ResourceInfoAndAccess
129     {
130         ResourceInfo          resourceInfo;
131         ResourceReadDelegate  readDelegate;  // May be invalid. In this case defaultReadDelegate is
      used.
132         ResourceWriteDelegate writeDelegate; // May be invalid. In this case defaultWriteDelegate is
      used.
133     };
134
141     IrisInstanceResource(IrisInstance* irisInstance = 0);
142
148     void attachTo(IrisInstance* irisInstance);
149
163     ResourceInfoAndAccess& addResource(const std::string& type,
164                                        const std::string& name,
165                                        const std::string& description);
166
179     void beginResourceGroup(const std::string& name,
180                             const std::string& description,
181                             uint64_t           startSubRscId = IRIS_UINT64_MAX,
182                             const std::string& cname         = std::string());
183
193     void setNextSubRscId(ResourceId nextSubRscId_)
194     {
195         nextSubRscId = nextSubRscId_;
196     }
197
206     void setTag(ResourceId rscId, const std::string& tag);
207
216     ResourceInfoAndAccess* getResourceInfo(ResourceId rscId);
217
238     static void calcHierarchicalNames(std::vector<ResourceInfo>& resourceInfos, const
      std::vector<ResourceGroupInfo>& groupInfos = std::vector<ResourceGroupInfo>());
239
```

```
254     static void makeNamesHierarchical(std::vector<ResourceInfo>& resourceInfos, const
       std::vector<ResourceGroupInfo>& groupInfos = std::vector<ResourceGroupInfo>());
255
256 protected:
257     // --- Iris function implementations ---
258
259     void impl_resource_getList(IrisReceivedRequest& request);
260
261     void impl_resource_getListOfResourceGroups(IrisReceivedRequest& request);
262
263     void impl_resource_getResourceInfo(IrisReceivedRequest& request);
264
265     void impl_resource_read(IrisReceivedRequest& request);
266
267     void impl_resource_write(IrisReceivedRequest& request);
268
269 private:
270
276     static void calcHierarchicalNamesInternal(std::vector<ResourceInfo>& resourceInfos, const
       std::map<ResourceId,size_t>& rscIdToIndex, std::vector<bool>& done, size_t index);
277
278     // --- State ---
279
281     IrisInstance* irisInstance;
282
284     IrisLogger log;
285
288     typedef std::vector<ResourceInfoAndAccess> ResourceInfoList;
289     ResourceInfoList                           resourceInfos;
290
292     typedef std::vector<ResourceGroupInfo> GroupInfoList;
293     GroupInfoList                          groupInfos;
294
296     typedef std::map<std::string, size_t> GroupNameToIndex;
297     GroupNameToIndex                      groupNameToIndex;
298
300     ResourceGroupInfo* currentAddGroup;
301
303     uint64_t nextSubRscId{IRIS_UINT64_MAX};
304 };
305
306 NAMESPACE_IRIS_END
307
308 #endif // #ifndef ARM_INCLUDE_IrisInstanceResource_source
```

9.39 IrisInstanceSemihosting.h File Reference

IrisInstance add-on to implement semihosting functionality.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include "iris/IrisInstanceEvent.h"
#include <mutex>
#include <queue>
```

Classes

- class iris::IrisInstanceSemihosting

9.39.1 Detailed Description

IrisInstance add-on to implement semihosting functionality.

Copyright

Copyright (C) 2017 Arm Limited. All rights reserved.

9.40 IrisInstanceSemihosting.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisInstanceSemihosting_h
```

```
9  #define ARM_INCLUDE_IrisInstanceSemihosting_h
10
11 #include "iris/detail/IrisCommon.h"
12 #include "iris/detail/IrisLogger.h"
13 #include "iris/detail/IrisObjects.h"
14
15 #include "iris/IrisInstanceEvent.h"
16
17 #include <mutex>
18 #include <queue>
19
20 NAMESPACE_IRIS_START
21
22 class IrisInstance;
23 class IrisInstanceEvent;
24 class IrisReceivedRequest;
25
26 namespace semihost
27 {
28
38     static const uint64_t COOKED = (0 << 0);
39
43     static const uint64_t RAW = (1 << 0);
44
50     static const uint64_t BLOCK = (0 << 1);
51
55     static const uint64_t NONBLOCK = (1 << 1);
56
60     static const uint64_t EMIT_EVENT = (0 << 2);
61
65     static const uint64_t NO_EVENT = (1 << 2);
66
70     static const uint64_t DEFAULT = COOKED | BLOCK | EMIT_EVENT;
71
79     static const uint64_t STDIN = 0;
80
84     static const uint64_t STDOUT = 1;
85
89     static const uint64_t STDERR = 2;
90
91 } // namespace semihost
92
93 class IrisInstanceSemihosting
94 {
95 private:
97     IrisInstance* iris_instance{nullptr};
98
101     IrisInstanceEvent* inst_event{nullptr};
102
104     std::map<uint64_t, unsigned> evSrcId_map{};
105
107     std::vector<IrisEventRegistry> event_registries{};
108
111     struct InputBuffer
112     {
113         std::queue<uint8_t> buffer;
114         bool empty_write{false};
115     };
116     std::map<uint64_t, InputBuffer> buffered_input_data{};
117
119     std::mutex buffer_mutex{};
120
122     std::mutex extension_mutex{};
123
124     uint64_t extension_retval{0};
125
126     IrisLogger log{};
127
129     std::atomic<bool> unblock_requested{false};
130
131     enum ExtensionState
132     {
133         XS_DISABLED,          // Semihosting extensions are not supported
134         XS_DORMANT,           // No ongoing semihosting extension call in progress
135         XS_WAITING_FOR_REPLY, // Event has been emitted, waiting for a reply for a client
136         XS_RETURNED,          // A client instance has called semihosting_return()
137         XS_NOT_IMPLEMENTED    // A client instance has called semihosting_notImplemented()
138
139     } extension_state{XS_DISABLED};
140
141 public:
142     IrisInstanceSemihosting(IrisInstance* iris_instance = nullptr, IrisInstanceEvent* inst_event =
    nullptr);
143
144     ~IrisInstanceSemihosting();
145
151     void attachTo(IrisInstance* iris_instance);
```

```
152
161     void setEventHandler(IrisInstanceEvent* handler);
162
177     std::vector<uint8_t> readData(uint64_t fDes, uint64_t max_size = 0, uint64_t flags =
        semihost::DEFAULT);
178
179     /*
180      * @brief Write data for a given file descriptor
181      *
182      * @param   fDes        File descriptor to write to. Usually semihost::STDOUT or semihost::STDERR.
183      * @param   data        Buffer containing the data to write.
184      * @param   size        Size of the data buffer in bytes.
185      * @return              Returns false if no client is registered for IRIS_SEMIHOSTING_OUTPUT events.
186      */
187     bool writeData(uint64_t fDes, const uint8_t* data, uint64_t size);
188
193     void enableExtensions();
194
209     std::pair<bool, uint64_t> semihostedCall(uint64_t operation, uint64_t parameter);
210
214     void unblock();
215
216 private:
218     void impl_semihosting_provideInputData(IrisReceivedRequest& request);
219
221     void impl_semihosting_return(IrisReceivedRequest& request);
222
224     void impl_semihosting_notImplemented(IrisReceivedRequest& request);
225
227     IrisErrorCode createEventStream(EventStream*& stream_out, const EventSourceInfo& info,
228                                     const std::vector<std::string>& requested_fields);
229
231     void notifyCall(uint64_t operation, uint64_t parameter);
232
233     class SemihostingEventStream;
234
235     IrisErrorCode enableEventStream(EventStream* stream, unsigned event_type);
236     IrisErrorCode disableEventStream(EventStream* stream, unsigned event_type);
237 };
238
239 NAMESPACE_IRIS_END
240
241 #endif // ARM_INCLUDE_IrisInstanceSemihosting_h
```

9.41 IrisInstanceSimulation.h File Reference

IrisInstance add-on to implement simulation_* functions.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include "iris/IrisInstantiationContext.h"
#include <map>
#include <mutex>
#include <string>
#include <vector>
```

Classes

- class iris::IrisInstanceSimulation

  An IrisInstance add-on that adds simulation functions for the SimulationEngine instance.
- class iris::IrisSimulationResetContext

  Provides context to a reset delegate call.

Typedefs

- typedef IrisDelegate< std::vector< ResourceInfo > & > iris::SimulationGetParameterInfoDelegate

  Delegate to get a list of parameter information.
- typedef IrisDelegate< InstantiationResult & > iris::SimulationInstantiateDelegate

  Delegate to instantiate the simulation.

- typedef IrisDelegate iris::SimulationRequestShutdownDelegate

  Delegate to request that the simulation be shut down.
- typedef IrisDelegate< const IrisSimulationResetContext & > iris::SimulationResetDelegate

  Delegate to reset the simulation.
- typedef IrisDelegate< const InstantiationParameterValue & > iris::SimulationSetParameterValueDelegate

  Delegate to set the value of an instantiation parameter.

Enumerations

- enum iris::IrisSimulationPhase {
  IRIS_SIM_PHASE_INITIAL_PLUGIN_LOADING_COMPLETE , **IRIS_SIM_PHASE_INSTANTIATE_ENTER** , **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 , **IRIS_SIM_PHASE_INITIAL_RESET_ENTER** , **IRIS_SIM_PHASE_INITIAL_RESET** , **IRIS_SIM_PHASE_INITIAL_RESET_LEAVE** ,
  IRIS_SIM_PHASE_START_OF_SIMULATION , **IRIS_SIM_PHASE_RESET_ENTER** , **IRIS_SIM_PHASE_RESET** , **IRIS_SIM_PHASE_RESET_LEAVE** ,
  IRIS_SIM_PHASE_END_OF_SIMULATION , **IRIS_SIM_PHASE_TERMINATE_ENTER** , **IRIS_SIM_PHASE_TERMINATE** , **IRIS_SIM_PHASE_TERMINATE_LEAVE** ,
  IRIS_SIM_PHASE_NUM }

  List of IRIS_SIMULATION_PHASE events.

9.41.1 Detailed Description

IrisInstance add-on to implement simulation_* functions.

Copyright

9.41.2 Typedef Documentation

9.41.2.1 SimulationGetParameterInfoDelegate

```
typedef IrisDelegate<std::vector<ResourceInfo>&> iris::SimulationGetParameterInfoDelegate
```

Delegate to get a list of parameter information.

```
IrisErrorCode getInstantiationParameterInfo(std::vector<ResourceInfo> &parameters_out)
```

9.41.2.2 SimulationInstantiateDelegate

```
typedef IrisDelegate<InstantiationResult&> iris::SimulationInstantiateDelegate
```

Delegate to instantiate the simulation.

```
IrisErrorCode instantiate(InstantiationResult &result_out)
```

9.41.2.3 SimulationRequestShutdownDelegate

```
typedef IrisDelegate iris::SimulationRequestShutdownDelegate
```

Delegate to request that the simulation be shut down.

```
IrisErrorCode requestShutdown()
```

9.41.2.4 SimulationResetDelegate

```
typedef IrisDelegate<const IrisSimulationResetContext&> iris::SimulationResetDelegate
```

Delegate to reset the simulation.

```
IrisErrorCode reset(const IrisSimulationResetContext &)
```

9.41.2.5 SimulationSetParameterValueDelegate

```
typedef IrisDelegate<const InstantiationParameterValue&> iris::SimulationSetParameterValueDelegate
```

Delegate to set the value of an instantiation parameter.

```
IrisErrorCode setInstantiationParameterValue(const InstantiationParameterValue &value)
```

9.42 IrisInstanceSimulation.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisInstanceSimulation_h
9 #define ARM_INCLUDE_IrisInstanceSimulation_h
10
11 #include "iris/detail/IrisCommon.h"
12 #include "iris/detail/IrisDelegate.h"
13 #include "iris/detail/IrisLogger.h"
14 #include "iris/detail/IrisObjects.h"
15
16 #include "iris/IrisInstantiationContext.h"
17
18 #include <map>
19 #include <mutex>
20 #include <string>
21 #include <vector>
22
23 NAMESPACE_IRIS_START
24
25 class IrisInstance;
26 class IrisReceivedRequest;
27 class IrisInstanceEvent;
28 class IrisEventRegistry;
29
30 class EventStream;
31
39 typedef IrisDelegate<InstantiationResult&> SimulationInstantiateDelegate;
40
44 class IrisSimulationResetContext
45 {
46 private:
48     static const uint64_t ALLOW_PARTIAL = (1 << 0);
49
51     uint64_t flags;
52
54     bool getFlag(uint64_t mask) const
55     {
56         return (flags & mask) != 0;
57     }
58
60     void setFlag(uint64_t mask, bool value)
61     {
62         flags &= ~mask;
63         flags |= (value ? mask : 0);
64     }
65
66 public:
67     IrisSimulationResetContext()
68         : flags(0)
69     {
70     }
71
77     bool getAllowPartialReset() const
78     {
79         return getFlag(ALLOW_PARTIAL);
80     }
81
82     // Set/clear the allowPartialReset flag.
83     void setAllowPartialReset(bool value = true)
84     {
85         setFlag(ALLOW_PARTIAL, value);
86     }
87 };
88
94 typedef IrisDelegate<const IrisSimulationResetContext&> SimulationResetDelegate;
```

```
95
101 typedef IrisDelegate<> SimulationRequestShutdownDelegate;
102
107 typedef IrisDelegate<std::vector<ResourceInfo>&> SimulationGetParameterInfoDelegate;
108
113 typedef IrisDelegate<const InstantiationParameterValue&> SimulationSetParameterValueDelegate;
114
118 enum IrisSimulationPhase
119 {
120     IRIS_SIM_PHASE_INITIAL_PLUGIN_LOADING_COMPLETE,
121     IRIS_SIM_PHASE_INSTANTIATE_ENTER,
122     IRIS_SIM_PHASE_INSTANTIATE,
123     IRIS_SIM_PHASE_INSTANTIATE_LEAVE,
124     IRIS_SIM_PHASE_INIT_ENTER,
125     IRIS_SIM_PHASE_INIT,
126     IRIS_SIM_PHASE_INIT_LEAVE,
127     IRIS_SIM_PHASE_BEFORE_END_OF_ELABORATION,
128     IRIS_SIM_PHASE_END_OF_ELABORATION,
129     IRIS_SIM_PHASE_INITIAL_RESET_ENTER,
130     IRIS_SIM_PHASE_INITIAL_RESET,
131     IRIS_SIM_PHASE_INITIAL_RESET_LEAVE,
132     IRIS_SIM_PHASE_START_OF_SIMULATION,
133     IRIS_SIM_PHASE_RESET_ENTER,
134     IRIS_SIM_PHASE_RESET,
135     IRIS_SIM_PHASE_RESET_LEAVE,
136     IRIS_SIM_PHASE_END_OF_SIMULATION,
137     IRIS_SIM_PHASE_TERMINATE_ENTER,
138     IRIS_SIM_PHASE_TERMINATE,
139     IRIS_SIM_PHASE_TERMINATE_LEAVE,
140     IRIS_SIM_PHASE_NUM
141 };
142 static const size_t IrisSimulationPhase_total = IRIS_SIM_PHASE_NUM;
143
147 class IrisInstanceSimulation
148 {
149 private:
151     IrisInstance* iris_instance;
152
155     IrisConnectionInterface* connection_interface;
156
158     SimulationInstantiateDelegate instantiate;
159
161     SimulationResetDelegate reset;
162
164     SimulationRequestShutdownDelegate requestShutdown;
165
167     SimulationGetParameterInfoDelegate getParameterInfo;
168
170     SimulationSetParameterValueDelegate setParameterValue;
171
174     enum
175     {
176         CACHE_DISABLED,
177         CACHE_EMPTY,
178         CACHE_SET
179     } parameter_info_cache_state;
180
182     std::vector<ResourceInfo> cached_parameter_info;
183
185     std::mutex mutex;
186
188     std::vector<IrisEventRegistry*> simulation_phase_event_registries;
189
191     std::map<uint64_t, IrisSimulationPhase> evSrcId_to_phase;
192
195     bool simulation_has_been_initialised;
196
198     std::vector<uint64_t> requests_waiting_for_instantiation;
199
201     unsigned logLevel{};
202
203 public:
211     IrisInstanceSimulation(IrisInstance*            iris_instance        = nullptr,
212                            IrisConnectionInterface* connection_interface = nullptr);
213     ~IrisInstanceSimulation();
214
220     void attachTo(IrisInstance* iris_instance);
221
227     void setConnectionInterface(IrisConnectionInterface* connection_interface_)
228     {
229         connection_interface = connection_interface_;
230     }
231
237     void setInstantiateDelegate(SimulationInstantiateDelegate delegate)
238     {
239         instantiate = delegate;
240     }
```

```
241
251     template <typename T, IrisErrorCode (T::*METHOD)(InstantiationResult&)>
252     void setInstantiateDelegate(T* instance)
253     {
254         setInstantiateDelegate(SimulationInstantiateDelegate::make<T, METHOD>(instance));
255     }
256
264     template <IrisErrorCode (*FUNC)(InstantiationResult&)>
265     void setInstantiateDelegate()
266     {
267         setInstantiateDelegate(SimulationInstantiateDelegate::make<FUNC>());
268     }
269
275     void setResetDelegate(SimulationResetDelegate delegate)
276     {
277         reset = delegate;
278     }
279
289     template <typename T, IrisErrorCode (T::*METHOD)(const IrisSimulationResetContext&)>
290     void setResetDelegate(T* instance)
291     {
292         setResetDelegate(SimulationResetDelegate::make<T, METHOD>(instance));
293     }
294
302     template <IrisErrorCode (*FUNC)(const IrisSimulationResetContext&)>
303     void setResetDelegate()
304     {
305         setResetDelegate(SimulationResetDelegate::make<FUNC>());
306     }
307
314     void setRequestShutdownDelegate(SimulationRequestShutdownDelegate delegate)
315     {
316         requestShutdown = delegate;
317     }
318
328     template <typename T, IrisErrorCode (T::*METHOD)()>
329     void setRequestShutdownDelegate(T* instance)
330     {
331         setRequestShutdownDelegate(SimulationRequestShutdownDelegate::make<T, METHOD>(instance));
332     }
333
341     template <IrisErrorCode (*FUNC)()>
342     void setRequestShutdownDelegate()
343     {
344         setRequestShutdownDelegate(SimulationRequestShutdownDelegate::make<FUNC>());
345     }
346
357     void setGetParameterInfoDelegate(SimulationGetParameterInfoDelegate delegate, bool cache_result =
    true)
358     {
359         getParameterInfo           = delegate;
360         parameter_info_cache_state = cache_result ? CACHE_EMPTY : CACHE_DISABLED;
361         cached_parameter_info.clear();
362     }
363
377     template <typename T, IrisErrorCode (T::*METHOD)(std::vector<ResourceInfo>&)>
378     void setGetParameterInfoDelegate(T* instance, bool cache_result = true)
379     {
380         typedef SimulationGetParameterInfoDelegate D;
381         setGetParameterInfoDelegate(D::make<T, METHOD>(instance), cache_result);
382     }
383
395     template <IrisErrorCode (*FUNC)(std::vector<ResourceInfo>&)>
396     void setGetParameterInfoDelegate(bool cache_result = true)
397     {
398         typedef SimulationGetParameterInfoDelegate D;
399         setGetParameterInfoDelegate(D::make<FUNC>(), cache_result);
400     }
401
408     void setSetParameterValueDelegate(SimulationSetParameterValueDelegate delegate)
409     {
410         setParameterValue = delegate;
411     }
412
422     template <typename T, IrisErrorCode (T::*METHOD)(const InstantiationParameterValue&)>
423     void setSetParameterValueDelegate(T* instance)
424     {
425         setSetParameterValueDelegate(SimulationSetParameterValueDelegate::make<T, METHOD>(instance));
426     }
427
435     template <IrisErrorCode (*FUNC)(const InstantiationParameterValue&)>
436     void setSetParameterValueDelegate()
437     {
438         setSetParameterValueDelegate(SimulationSetParameterValueDelegate::make<FUNC>());
439     }
440
449     void enterPostInstantiationPhase();
```

```
450
456     void setEventHandler(IrisInstanceEvent* handler);
457
465     void notifySimPhase(uint64_t time, IrisSimulationPhase phase, const IrisValueMap *fields = nullptr);
466
478     void registerSimEventsOnGlobalInstance();
479
485     static std::string getSimulationPhaseName(IrisSimulationPhase phase);
486
492     static std::string getSimulationPhaseDescription(IrisSimulationPhase phase);
493
499     void setLogLevel(unsigned logLevel_);
500
501 private:
503     void impl_simulation_getInstantiationParameterInfo(IrisReceivedRequest& request);
504
506     void impl_simulation_setInstantiationParameterValues(IrisReceivedRequest& request);
507
509     void impl_simulation_instantiate(IrisReceivedRequest& request);
510
512     void impl_simulation_reset(IrisReceivedRequest& request);
513
515     void impl_simulation_requestShutdown(IrisReceivedRequest& request);
516
518     void impl_simulation_waitForInstantiation(IrisReceivedRequest& request);
519
521     IrisErrorCode createEventStream(EventStream*& event_stream_out, const EventSourceInfo& info,
522                                     const std::vector<std::string>& fields);
523 };
524
525 NAMESPACE_IRIS_END
526
527 #endif // ARM_INCLUDE_IrisInstanceSimulation_h
```

9.43 IrisInstanceSimulationTime.h File Reference

IrisInstance add-on to implement simulationTime functions.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include <string>
#include <vector>
#include <functional>
```

Classes

- class iris::IrisInstanceSimulationTime

  Simulation time add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< uint64_t &, uint64_t &, bool & > iris::SimulationTimeGetDelegate

  Delegate to get the simulation time.
- typedef IrisDelegate iris::SimulationTimeRunDelegate

  Delegate to resume the simulation time progress.
- typedef IrisDelegate iris::SimulationTimeStopDelegate

  Delegate to stop the simulation time progress.

Enumerations

- enum iris::TIME_EVENT_REASON {
  iris::TIME_EVENT_NO_REASON = 0 , iris::TIME_EVENT_UNKNOWN = (1 << 0) , iris::TIME_EVENT_STOP
  = (1 << 1) , iris::TIME_EVENT_BREAKPOINT = (1 << 2) ,
  iris::TIME_EVENT_EVENT_COUNTER_OVERFLOW = (1 << 3) , iris::TIME_EVENT_STEPPING_COMPLETED
  = (1 << 4) , iris::TIME_EVENT_REACHED_DEBUGGABLE_STATE = (1 << 5) , iris::TIME_EVENT_EVENT
  = (1 << 6) ,
  iris::TIME_EVENT_STATE_CHANGED = (1 << 7) }

The reasons why the simulation time stopped. Bit masks.

9.43.1 Detailed Description

IrisInstance add-on to implement simulationTime functions.

Copyright

9.43.2 Typedef Documentation

9.43.2.1 SimulationTimeGetDelegate

```
typedef IrisDelegate<uint64_t&, uint64_t&, bool&> iris::SimulationTimeGetDelegate
```

Delegate to get the simulation time.

```
IrisErrorCode getTime(uint64_t &ticks, uint64_t &tickHz, bool &running);
```

9.43.2.2 SimulationTimeRunDelegate

```
typedef IrisDelegate iris::SimulationTimeRunDelegate
```

Delegate to resume the simulation time progress.

```
IrisErrorCode run();
```

9.43.2.3 SimulationTimeStopDelegate

```
typedef IrisDelegate iris::SimulationTimeStopDelegate
```

Delegate to stop the simulation time progress.

```
IrisErrorCode stop();
```

9.43.3 Enumeration Type Documentation

9.43.3.1 TIME_EVENT_REASON

```
enum iris::TIME_EVENT_REASON
```

The reasons why the simulation time stopped. Bit masks.
Note that Fast Models only ever emits TIME_EVENT_UNKNOWN.

Enumerator

TIME_EVENT_NO_REASON	Do not emit a REASON field.
TIME_EVENT_UNKNOWN	Simulation stopped for any reason.
TIME_EVENT_STOP	simulationTime_stop() was called.
TIME_EVENT_BREAKPOINT	Breakpoint was hit.
TIME_EVENT_EVENT_COUNTER_OVERFLOW	EventCounterMode.overflowStopSim.
TIME_EVENT_STEPPING_COMPLETED	step_setup() and then simulationTime_run().
TIME_EVENT_REACHED_DEBUGGABLE_STATE	simulationTime_runUntilDebuggableState().
TIME_EVENT_EVENT	eventStream_create(stop=true).
TIME_EVENT_STATE_CHANGED	State of any component changed.

9.44 IrisInstanceSimulationTime.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisInstanceSimulationTime_h
9 #define ARM_INCLUDE_IrisInstanceSimulationTime_h
10
11 #include "iris/detail/IrisCommon.h"
12 #include "iris/detail/IrisDelegate.h"
13
14 #include <string>
15 #include <vector>
16 #include <functional>
17
18 NAMESPACE_IRIS_START
19
24 typedef IrisDelegate<> SimulationTimeRunDelegate;
25
30 typedef IrisDelegate<> SimulationTimeStopDelegate;
31
36 typedef IrisDelegate<uint64_t&, uint64_t&, bool&> SimulationTimeGetDelegate;
37
43 enum TIME_EVENT_REASON
44 {
45     TIME_EVENT_NO_REASON = 0,
46     TIME_EVENT_UNKNOWN = (1 « 0),
47     TIME_EVENT_STOP = (1 « 1),
48     TIME_EVENT_BREAKPOINT = (1 « 2),
49     TIME_EVENT_EVENT_COUNTER_OVERFLOW = (1 « 3),
50     TIME_EVENT_STEPPING_COMPLETED = (1 « 4),
51     TIME_EVENT_REACHED_DEBUGGABLE_STATE = (1 « 5),
52     TIME_EVENT_EVENT = (1 « 6),
53     TIME_EVENT_STATE_CHANGED = (1 « 7),
54 };
55
56 class IrisInstance;
57 class IrisInstanceEvent;
58 class IrisEventRegistry;
59 class IrisReceivedRequest;
60
61 class EventStream;
62 struct EventSourceInfo;
63
67 class IrisInstanceSimulationTime
68 {
69 private:
71     IrisInstance* iris_instance;
72
74     IrisEventRegistry* simulation_time_event_registry;
75
77     SimulationTimeRunDelegate  run_delegate;
78     SimulationTimeStopDelegate stop_delegate;
79     SimulationTimeGetDelegate  get_time_delegate;
80     std::function<void()> notify_state_changed_delegate;
81
82 public:
90     IrisInstanceSimulationTime(IrisInstance* iris_instance = nullptr, IrisInstanceEvent* inst_event =
      nullptr);
91     ~IrisInstanceSimulationTime();
92
98     void attachTo(IrisInstance* irisInstance);
99
106     void setEventHandler(IrisInstanceEvent* handler);
107
113     void setSimTimeRunDelegate(SimulationTimeRunDelegate delegate)
114     {
115         run_delegate = delegate;
116     }
117
125     template <typename T, IrisErrorCode (T::*METHOD)()>
126     void setSimTimeRunDelegate(T* instance)
127     {
128         setSimTimeRunDelegate(SimulationTimeRunDelegate::make<T, METHOD>(instance));
129     }
130
138     template <IrisErrorCode (*FUNC)()>
139     void setSimTimeRunDelegate()
140     {
141         setSimTimeRunDelegate(SimulationTimeRunDelegate::make<FUNC>());
142     }
143
149     void setSimTimeStopDelegate(SimulationTimeStopDelegate delegate)
150     {
151         stop_delegate = delegate;
152     }
153
```

```
161     template <typename T, IrisErrorCode (T::*METHOD)()>
162     void setSimTimeStopDelegate(T* instance)
163     {
164         setSimTimeStopDelegate(SimulationTimeStopDelegate::make<T, METHOD>(instance));
165     }
166
174     template <IrisErrorCode (*FUNC)()>
175     void setSimTimeStopDelegate()
176     {
177         setSimTimeStopDelegate(SimulationTimeStopDelegate::make<FUNC>());
178     }
179
185     void setSimTimeGetDelegate(SimulationTimeGetDelegate delegate)
186     {
187         get_time_delegate = delegate;
188     }
189
197     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t&, uint64_t&, bool&)>
198     void setSimTimeGetDelegate(T* instance)
199     {
200         setSimTimeGetDelegate(SimulationTimeGetDelegate::make<T, METHOD>(instance));
201     }
202
210     template <IrisErrorCode (*FUNC)(uint64_t&, uint64_t&, bool&)>
211     void setSimTimeGetDelegate()
212     {
213         setSimTimeGetDelegate(SimulationTimeGetDelegate::make<FUNC>());
214     }
215
233     void setSimTimeNotifyStateChanged(std::function<void()> func)
234     {
235         notify_state_changed_delegate = func;
236     }
237
239     void notifySimulationTimeEvent(uint64_t reason = TIME_EVENT_UNKNOWN);
240
252     void registerSimTimeEventsOnGlobalInstance();
253
254 private:
256     void impl_simulationTime_run(IrisReceivedRequest& request);
257     void impl_simulationTime_stop(IrisReceivedRequest& request);
258     void impl_simulationTime_get(IrisReceivedRequest& request);
259     void impl_simulationTime_notifyStateChanged(IrisReceivedRequest& request);
260
261     IrisErrorCode createEventStream(EventStream*&, const EventSourceInfo&, const
        std::vector<std::string>&);
262 };
263
264 NAMESPACE_IRIS_END
265
266 #endif // ARM_INCLUDE_IrisInstanceSimulationTime_h
```

9.45 IrisInstanceStep.h File Reference

Stepping-related add-on to an IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisLogger.h"
#include "iris/detail/IrisObjects.h"
#include <cstdio>
```

Classes

- class iris::IrisInstanceStep

  Step add-on for IrisInstance.

Typedefs

- typedef IrisDelegate< uint64_t &, const std::string & > iris::RemainingStepGetDelegate

  Delegate to get the value of the currently remaining steps.

- typedef IrisDelegate< uint64_t, const std::string & > iris::RemainingStepSetDelegate

  Delegate to set the remaining steps measured in the specified unit.

- typedef IrisDelegate< uint64_t &, const std::string & > iris::StepCountGetDelegate

  Delegate to get the value of the step count.

9.45.1 Detailed Description

Stepping-related add-on to an IrisInstance.

Copyright

Copyright (C) 2016 Arm Limited. All rights reserved.

The IrisInstanceStep class implements all stepping-related Iris functions.

9.45.2 Typedef Documentation

9.45.2.1 RemainingStepGetDelegate

```
typedef IrisDelegate<uint64_t&, const std::string&> iris::RemainingStepGetDelegate
```

Delegate to get the value of the currently remaining steps.

```
IrisErrorCode getRemainingSteps(uint64_t &steps, const std::string &unit)
```

Error: Return E_* error code if it failed to get the remaining steps.

9.45.2.2 RemainingStepSetDelegate

```
typedef IrisDelegate<uint64_t, const std::string&> iris::RemainingStepSetDelegate
```

Delegate to set the remaining steps measured in the specified unit.

```
IrisErrorCode setRemainingSteps(uint64_t steps, const std::string &unit)
```

Error: Return E_* error code if it failed to set the steps.

9.45.2.3 StepCountGetDelegate

```
typedef IrisDelegate<uint64_t&, const std::string&> iris::StepCountGetDelegate
```

Delegate to get the value of the step count.

```
IrisErrorCode getStepCount(uint64_t &count, const std::string &unit)
```

Error: Return E_* error code if it failed to get the step count.

9.46 IrisInstanceStep.h

Go to the documentation of this file.

```
1
9 #ifndef ARM_INCLUDE_IrisInstanceStep_h
10 #define ARM_INCLUDE_IrisInstanceStep_h
11
12 #include "iris/detail/IrisCommon.h"
13 #include "iris/detail/IrisDelegate.h"
14 #include "iris/detail/IrisLogger.h"
15 #include "iris/detail/IrisObjects.h"
16
17 #include <cstdio>
18
19 NAMESPACE_IRIS_START
20
21 class IrisInstance;
22 class IrisReceivedRequest;
23
31 typedef IrisDelegate<uint64_t, const std::string&> RemainingStepSetDelegate;
32
40 typedef IrisDelegate<uint64_t&, const std::string&> RemainingStepGetDelegate;
41
49 typedef IrisDelegate<uint64_t&, const std::string&> StepCountGetDelegate;
50
58 class IrisInstanceStep
59 {
60 public:
66     IrisInstanceStep(IrisInstance* irisInstance = nullptr);
67
75     void attachTo(IrisInstance* irisInstance);
```

```
76
83     void setRemainingStepSetDelegate(RemainingStepSetDelegate delegate);
84
91     void setRemainingStepGetDelegate(RemainingStepGetDelegate delegate);
92
99     void setStepCountGetDelegate(StepCountGetDelegate delegate);
100
101 private:
102     void impl_step_setup(IrisReceivedRequest& request);
103
104     void impl_step_getRemainingSteps(IrisReceivedRequest& request);
105
106     void impl_step_getStepCounterValue(IrisReceivedRequest& request);
107
108     void impl_step_syncStep(IrisReceivedRequest& request);
109
110     void impl_step_syncStepSetup(IrisReceivedRequest& request);
111
113
115     IrisInstance* irisInstance;
116
118     RemainingStepSetDelegate stepSetDel;
119     RemainingStepGetDelegate stepGetDel;
120
122     StepCountGetDelegate stepCountGetDel;
123
125     IrisLogger log;
126 };
127
128 NAMESPACE_IRIS_END
129
130 #endif // #ifndef ARM_INCLUDE_IrisInstanceStep_h
```

9.47 IrisInstanceTable.h File Reference

Table add-on to IrisInstance.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisDelegate.h"
#include "iris/detail/IrisObjects.h"
```

Classes

- class iris::IrisInstanceTable

  Table add-on for IrisInstance.
- struct iris::IrisInstanceTable::TableInfoAndAccess

  Entry in 'tableInfos'.

Typedefs

- typedef IrisDelegate< const TableInfo &, uint64_t, uint64_t, TableReadResult & > iris::TableReadDelegate

  Delegate to read table data.
- typedef IrisDelegate< const TableInfo &, const TableRecords &, TableWriteResult & > iris::TableWriteDelegate

  Delegate to write table data.

9.47.1 Detailed Description

Table add-on to IrisInstance.

Copyright

Copyright (C) 2016 Arm Limited. All rights reserved.

The IrisInstanceTable class implements all table-related Iris functions.

9.47.2 Typedef Documentation

9.47.2.1 TableReadDelegate

```
typedef IrisDelegate<const TableInfo&, uint64_t, uint64_t, TableReadResult&> iris::TableReadDelegate
```

Delegate to read table data.

```
IrisErrorCode read(const TableInfo &tableInfo, uint64_t index, uint64_t count, TableReadResult &result)
```

tableInfo, index, and count are guaranteed to be valid. count is non-zero.
TableReadResult holds the read results and any errors from reading table cell values.

9.47.2.2 TableWriteDelegate

```
typedef IrisDelegate<const TableInfo&, const TableRecords&, TableWriteResult&> iris::TableWriteDelegate
```

Delegate to write table data.

```
IrisErrorCode write(const TableInfo &tableInfo, const TableRecords &records, TableWriteResult &result)
```

records is guaranteed to be non-empty.
TableWriteResult holds any errors from writing table cell values.

9.48 IrisInstanceTable.h

Go to the documentation of this file.

```
1
9  #ifndef ARM_INCLUDE_IrisInstanceTable_h
10 #define ARM_INCLUDE_IrisInstanceTable_h
11
12 #include "iris/detail/IrisCommon.h"
13 #include "iris/detail/IrisDelegate.h"
14 #include "iris/detail/IrisObjects.h"
15
16 NAMESPACE_IRIS_START
17
18 class IrisInstance;
19 class IrisReceivedRequest;
20
31 typedef IrisDelegate<const TableInfo&, uint64_t, uint64_t, TableReadResult&> TableReadDelegate;
32
43 typedef IrisDelegate<const TableInfo&, const TableRecords&, TableWriteResult&> TableWriteDelegate;
44
50 class IrisInstanceTable
51 {
52 public:
58     struct TableInfoAndAccess
59     {
60         TableInfo          tableInfo;
61         TableReadDelegate  readDelegate;
62         TableWriteDelegate writeDelegate;
63     };
64
70     IrisInstanceTable(IrisInstance* irisInstance = nullptr);
71
79     void attachTo(IrisInstance* irisInstance);
80
88     TableInfoAndAccess& addTableInfo(const std::string& name);
89
96     void setDefaultReadDelegate(TableReadDelegate delegate = TableReadDelegate())
97     {
98         defaultReadDelegate = delegate;
99     }
100
107     void setDefaultWriteDelegate(TableWriteDelegate delegate = TableWriteDelegate())
108     {
109         defaultWriteDelegate = delegate;
110     }
111
112 private:
113     void impl_table_getList(IrisReceivedRequest& request);
114
115     void impl_table_read(IrisReceivedRequest& request);
116
117     void impl_table_write(IrisReceivedRequest& request);
118
120
122     IrisInstance* irisInstance;
123
125     typedef std::vector<TableInfoAndAccess> TableInfoAndAccessList;
126     TableInfoAndAccessList                  tableInfos;
127
129     TableReadDelegate  defaultReadDelegate;
130     TableWriteDelegate defaultWriteDelegate;
131 };
132
```

```
133 NAMESPACE_IRIS_END
134
135 #endif // #ifndef ARM_INCLUDE_IrisInstanceTable_h
```

9.49 IrisInstantiationContext.h File Reference

Helper class used to instantiate Iris instances from generic factories.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisObjects.h"
#include "iris/detail/IrisUtils.h"
#include <string>
#include <vector>
```

Classes

- class iris::IrisInstantiationContext

  Provides context when instantiating an Iris instance from a factory.

9.49.1 Detailed Description

Helper class used to instantiate Iris instances from generic factories.

Copyright

Copyright (C) 2017-2023 Arm Limited. All rights reserved.

9.50 IrisInstantiationContext.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisInstantiationContext_h
8 #define ARM_INCLUDE_IrisInstantiationContext_h
9
10 #include "iris/detail/IrisCommon.h"
11 #include "iris/detail/IrisObjects.h"
12 #include "iris/detail/IrisUtils.h"
13
14 #include <string>
15 #include <vector>
16
17 NAMESPACE_IRIS_START
18
22 class IrisInstantiationContext
23 {
24 private:
25     IrisConnectionInterface* connection_interface;
26
29     InstantiationResult& result;
30
33     IrisValueMap params;
34
39     std::string prefix;
40
42     std::string component_name;
43
44     uint64_t instance_flags;
45
47     std::vector<IrisInstantiationContext*> children;
48
49     void errorInternal(const std::string& severity,
50                        const std::string& code,
51                        const std::string& parameterName,
52                        const char*        format,
53                        va_list            args);
54
57     void processParameters(const std::vector<ResourceInfo>&                param_info_,
58                            const std::vector<InstantiationParameterValue>& param_values_);
59
61     IrisInstantiationContext(const IrisInstantiationContext* parent, const std::string& instance_name);
62
63 public:
```

```
64      IrisInstantiationContext(IrisConnectionInterface*                       connection_interface_,
65                               InstantiationResult&                           result_,
66                               const std::vector<ResourceInfo>&               param_info_,
67                               const std::vector<InstantiationParameterValue>& param_values_,
68                               const std::string&                             prefix_,
69                               const std::string&                             component_name_,
70                               uint64_t                                       instance_flags_);
71
72      ~IrisInstantiationContext();
73
85      IrisInstantiationContext* getSubcomponentContext(const std::string& child_name);
86
96      template <typename T>
97      void getParameter(const std::string& name, T& value)
98      {
99          getParameter(name).get(value);
100     }
101
111     const IrisValue& getParameter(const std::string& name)
112     {
113         IrisValueMap::const_iterator it = params.find(name);
114         if (it == params.end())
115         {
116             throw IrisInternalError("getParameter(" + name + "): Unknown parameter");
117         }
118         return it->second;
119     }
120
127     std::string getStringParameter(const std::string& name)
128     {
129         return getParameter(name).getAsString();
130     }
131
138     uint64_t getU64Parameter(const std::string& name)
139     {
140         return getParameter(name).getAsU64();
141     }
142
149     int64_t getS64Parameter(const std::string& name)
150     {
151         return getParameter(name).getAsS64();
152     }
153
160     bool getBoolParameter(const std::string& name)
161     {
162         return getParameter(name).getAsBool();
163     }
164
174     void getParameter(const std::string& name, std::vector<uint64_t>& value);
175
182     uint64_t getRecommendedInstanceFlags() const
183     {
184         return instance_flags;
185     }
186
193     std::string getInstanceName() const
194     {
195         return prefix + "." + component_name;
196     }
197
203     IrisConnectionInterface* getConnectionInterface() const
204     {
205         return connection_interface;
206     }
207
218     void warning(const std::string& code, const char* format, ...) INTERNAL_IRIS_PRINTF(3, 4);
219
231     void parameterWarning(const std::string& code, const std::string& parameterName, const char* format,
        ...) INTERNAL_IRIS_PRINTF(4, 5);
242     void error(const std::string& code, const char* format, ...) INTERNAL_IRIS_PRINTF(3, 4);
243
255     void parameterError(const std::string& code, const std::string& parameterName, const char* format,
        ...) INTERNAL_IRIS_PRINTF(4, 5);
256 };
257
258 NAMESPACE_IRIS_END
259
260 #endif // ARM_INCLUDE_IrisInstantiationContext_h
```

9.51 IrisParameterBuilder.h File Reference

Helper class to construct instantiation parameters.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisObjects.h"
#include <string>
#include <vector>
```

Classes

- class iris::IrisParameterBuilder

  Helper class to construct instantiation parameters.

9.51.1 Detailed Description

Helper class to construct instantiation parameters.

Copyright

9.52 IrisParameterBuilder.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisParameterBuilder_h
8 #define ARM_INCLUDE_IrisParameterBuilder_h
9
10 #include "iris/detail/IrisCommon.h"
11 #include "iris/detail/IrisObjects.h"
12
13 #include <string>
14 #include <vector>
15
16 NAMESPACE_IRIS_START
17
21 class IrisParameterBuilder
22 {
23 private:
24     ResourceInfo& info;
25
26     IrisParameterBuilder& setValueExtend(std::vector<uint64_t>& arr, uint64_t value, uint64_t extension)
27     {
28         arr.resize(info.getDataSizeInU64Chunks(), extension);
29         arr[0] = value;
30
31         return *this;
32     }
33
34     IrisParameterBuilder& setValueExtend(std::vector<uint64_t>& arr, const std::vector<uint64_t>& value,
      uint64_t extension)
35     {
36         size_t param_size = info.getDataSizeInU64Chunks();
37         if (param_size < value.size())
38         {
39             throw IrisInternalError("Invalid parameter configuration");
40         }
41         arr = value;
42         arr.resize(info.getDataSizeInU64Chunks(), extension);
43
44         return *this;
45     }
46
47     IrisParameterBuilder& setValueSignExtend(std::vector<uint64_t>& arr, int64_t value)
48     {
49         return setValueExtend(arr, static_cast<uint64_t>(value), (value < 0) ? IRIS_UINT64_MAX : 0);
50     }
51
52     IrisParameterBuilder& setValueZeroExtend(std::vector<uint64_t>& arr, uint64_t value)
53     {
54         return setValueExtend(arr, value, 0);
55     }
56
57     IrisParameterBuilder& setValueSignExtend(std::vector<uint64_t>& arr, const std::vector<uint64_t>&
      value)
58     {
59         return setValueExtend(arr, value, (static_cast<int64_t>(value.back()) < 0) ? IRIS_UINT64_MAX :
      0);
```

```
60      }
61
62      IrisParameterBuilder& setValueZeroExtend(std::vector<uint64_t>& arr, const std::vector<uint64_t>&
        value)
63      {
64          return setValueExtend(arr, value, 0);
65      }
66
67      IrisParameterBuilder& setValueDouble(std::vector<uint64_t>& arr, double value)
68      {
69          arr.resize(1);
70          *static_cast<double*>((void*) (&arr[0])) = value;
71
72          return *this;
73      }
74
75  public:
80      IrisParameterBuilder(ResourceInfo& info_)
81          : info(info_)
82      {
83          info.isParameter = true;
84      }
85
91      IrisParameterBuilder& setName(const std::string& name)
92      {
93          info.name = name;
94          return *this;
95      }
96
102     IrisParameterBuilder& setDescr(const std::string& description)
103     {
104         info.description = description;
105         return *this;
106     }
107
113     IrisParameterBuilder& setFormat(const std::string& format)
114     {
115         info.format = format;
116         return *this;
117     }
118
124     IrisParameterBuilder& setBitWidth(uint64_t bitWidth)
125     {
126         info.bitWidth = bitWidth;
127         return *this;
128     }
129
135     IrisParameterBuilder& setRwMode(const std::string& rwMode)
136     {
137         info.rwMode = rwMode;
138         return *this;
139     }
140
146     IrisParameterBuilder& setSubRscId(uint64_t subRscId)
147     {
148         info.subRscId = subRscId;
149         return *this;
150     }
151
157     IrisParameterBuilder& setTopology(bool value = true)
158     {
159         info.parameterInfo.topology = value;
160         return *this;
161     }
162
168     IrisParameterBuilder& setInitOnly(bool value = true)
169     {
170         info.parameterInfo.initOnly = value;
171         return *this;
172     }
173
179     IrisParameterBuilder& setMin(uint64_t min)
180     {
181         return setValueZeroExtend(info.parameterInfo.min, min);
182     }
183
189     IrisParameterBuilder& setMax(uint64_t max)
190     {
191         return setValueZeroExtend(info.parameterInfo.max, max);
192     }
193
200     IrisParameterBuilder& setRange(uint64_t min, uint64_t max)
201     {
202         return setMin(min).setMax(max);
203     }
204
213     IrisParameterBuilder& setMin(const std::vector<uint64_t>& min)
```

```
214     {
215         return setValueZeroExtend(info.parameterInfo.min, min);
216     }
217
226     IrisParameterBuilder& setMax(const std::vector<uint64_t>& max)
227     {
228         return setValueZeroExtend(info.parameterInfo.max, max);
229     }
230
240     IrisParameterBuilder& setRange(const std::vector<uint64_t>& min, const std::vector<uint64_t>& max)
241     {
242         return setMin(min).setMax(max);
243     }
244
253     IrisParameterBuilder& setMinSigned(int64_t min)
254     {
255         return setValueSignExtend(info.parameterInfo.min, min)
256             .setType("numericSigned");
257     }
258
267     IrisParameterBuilder& setMaxSigned(int64_t max)
268     {
269         return setValueSignExtend(info.parameterInfo.max, max)
270             .setType("numericSigned");
271     }
272
282     IrisParameterBuilder& setRangeSigned(int64_t min, int64_t max)
283     {
284         return setValueSignExtend(info.parameterInfo.min, min)
285             .setValueSignExtend(info.parameterInfo.max, max)
286             .setType("numericSigned");
287     }
288
298     IrisParameterBuilder& setMinSigned(const std::vector<uint64_t>& min)
299     {
300         return setValueSignExtend(info.parameterInfo.min, min)
301             .setType("numericSigned");
302     }
303
313     IrisParameterBuilder& setMaxSigned(const std::vector<uint64_t>& max)
314     {
315         return setValueSignExtend(info.parameterInfo.max, max)
316             .setType("numericSigned");
317     }
318
329     IrisParameterBuilder& setRangeSigned(const std::vector<uint64_t>& min, const std::vector<uint64_t>&
    max)
330     {
331         return setValueSignExtend(info.parameterInfo.min, min)
332             .setValueSignExtend(info.parameterInfo.max, max)
333             .setType("numericSigned");
334     }
335
344     IrisParameterBuilder& setMinFloat(double min)
345     {
346         return setValueDouble(info.parameterInfo.min, min)
347             .setType("numericFp");
348     }
349
358     IrisParameterBuilder& setMaxFloat(double max)
359     {
360         return setValueDouble(info.parameterInfo.max, max)
361             .setType("numericFp");
362     }
363
373     IrisParameterBuilder& setRangeFloat(double min, double max)
374     {
375         return setValueDouble(info.parameterInfo.min, min)
376             .setValueDouble(info.parameterInfo.max, max)
377             .setType("numericFp");
378     }
379
388     IrisParameterBuilder& addEnum(const std::string& symbol, const IrisValue& value, const std::string&
    description = std::string())
389     {
390         info.enums.push_back(EnumElementInfo(value, symbol, description));
391         return *this;
392     }
393
403     IrisParameterBuilder& addStringEnum(const std::string& value, const std::string& description =
    std::string())
404     {
405         info.enums.push_back(EnumElementInfo(IrisValue(value), std::string(), description));
406         return *this;
407     }
408
415     IrisParameterBuilder& setTag(const std::string& tag)
```

```
416     {
417         info.tags[tag] = IrisValue(true);
418         return *this;
419     }
420
427     IrisParameterBuilder& setHidden(bool hidden)
428     {
429         info.isHidden = hidden;
430         return *this;
431     }
432
440     IrisParameterBuilder& setTag(const std::string& tag, const IrisValue& value)
441     {
442         info.tags[tag] = value;
443         return *this;
444     }
445
452     IrisParameterBuilder& setDefault(const std::string& value)
453     {
454         info.parameterInfo.defaultString = value;
455         return *this;
456     }
457
464     IrisParameterBuilder& setDefault(uint64_t value)
465     {
466         return setValueZeroExtend(info.parameterInfo.defaultData, value);
467     }
468
477     IrisParameterBuilder& setDefault(const std::vector<uint64_t>& value)
478     {
479         return setValueZeroExtend(info.parameterInfo.defaultData, value);
480     }
481
488     IrisParameterBuilder& setDefaultSigned(int64_t value)
489     {
490         return setValueSignExtend(info.parameterInfo.defaultData, value);
491     }
492
501     IrisParameterBuilder& setDefaultSigned(const std::vector<uint64_t>& value)
502     {
503         return setValueSignExtend(info.parameterInfo.defaultData, value);
504     }
505
512     IrisParameterBuilder& setDefaultFloat(double value)
513     {
514         return setValueDouble(info.parameterInfo.defaultData, value);
515     }
516
525     IrisParameterBuilder& setType(const std::string& type)
526     {
527         if ((info.bitWidth != 32) && (info.bitWidth != 64) && (type == "numericFp"))
528         {
529             throw IrisInternalError(
530                 "Invalid parameter configuration."
531                 " NumericFp parameters must have a bitWidth of 32 or 64");
532         }
533
534         info.type = type;
535         return *this;
536     }
537 };
538
539 NAMESPACE_IRIS_END
540
541 #endif // ARM_INCLUDE_IrisParameterBuilder_h
```

9.53 IrisPluginFactory.h File Reference

A generic plug-in factory for instantiating plug-in instances.

```
#include "iris/IrisCConnection.h"
#include "iris/IrisInstance.h"
#include "iris/IrisInstanceFactoryBuilder.h"
#include "iris/IrisInstantiationContext.h"
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisFunctionInfo.h"
#include "iris/detail/IrisObjects.h"
#include "iris/detail/IrisU64JsonReader.h"
#include "iris/detail/IrisU64JsonWriter.h"
#include <mutex>
```

```
#include <string>
#include <vector>
```

Classes

- class iris::IrisNonFactoryPlugin< PLUGIN_CLASS >

  Wrapper to instantiate a non-factory plugin.
- class iris::IrisPluginFactory< PLUGIN_CLASS >
- class iris::IrisPluginFactoryBuilder

  Set meta data for instantiating a plug-in instance.

Macros

- #define IRIS_NON_FACTORY_PLUGIN(PluginClassName)

  Create plugin entry point for non-factory plugins (i.e. plugins which do not have parameters and which are always instantiated just once).
- #define IRIS_PLUGIN_FACTORY(PluginClassName)

  Create plugin entry point for plugins which have a factory (i.e. plugins which have parameters and/or plugins which are potentially instantiated multiple times).

9.53.1 Detailed Description

A generic plug-in factory for instantiating plug-in instances.

Copyright

9.53.2 Macro Definition Documentation

9.53.2.1 IRIS_NON_FACTORY_PLUGIN

```
#define IRIS_NON_FACTORY_PLUGIN(
              PluginClassName )
```

Value:

```
extern "C" IRIS_EXPORT int64_t irisInitPlugin(IrisC_Functions* functions)                          \
{                                                                                                  \
    return ::iris::IrisNonFactoryPlugin<PluginClassName>::initPlugin(functions, #PluginClassName);   \
}
```

Create plugin entry point for non-factory plugins (i.e. plugins which do not have parameters and which are always instantiated just once).

Parameters

PluginClassName	Class name of the plugin.

9.53.2.2 IRIS_PLUGIN_FACTORY

```
#define IRIS_PLUGIN_FACTORY(
              PluginClassName )
```

Value:

```
    extern "C" IRIS_EXPORT int64_t irisInitPlugin(IrisC_Functions* functions)                      \
    {                                                                                              \
        return ::iris::IrisPluginFactory<PluginClassName>::initPlugin(functions, #PluginClassName); \
    }
```

Create plugin entry point for plugins which have a factory (i.e. plugins which have parameters and/or plugins which are potentially instantiated multiple times).

Parameters

PluginClassName	Objects of this type are instantiated for each plug-in instance created.

9.54 IrisPluginFactory.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisPluginFactory_h
8 #define ARM_INCLUDE_IrisPluginFactory_h
9
10 #include "iris/IrisCConnection.h"
11 #include "iris/IrisInstance.h"
12 #include "iris/IrisInstanceFactoryBuilder.h"
13 #include "iris/IrisInstantiationContext.h"
14 #include "iris/detail/IrisCommon.h"
15 #include "iris/detail/IrisFunctionInfo.h"
16 #include "iris/detail/IrisObjects.h"
17 #include "iris/detail/IrisU64JsonReader.h"
18 #include "iris/detail/IrisU64JsonWriter.h"
19
20 #include <mutex>
21 #include <string>
22 #include <vector>
23
24 NAMESPACE_IRIS_START
25
26 // Iris plugins
27 // ============
28 //
29 // This header supports declaring two different kind of plugins by using one of two macros:
30 //
31 // 1. Factory plugins:
32 //
33 // IRIS_PLUGIN_FACTORY(PluginClassName)
34 //
35 // where PluginClassName is the class of the plugin, not the factory. The factory is instantiated
       automatically by the macro.
36 //
37 // This declares a plugin which has a plugin factory. This type of plugin must be used
38 // for plugins which have parameters and for plugins where it makes sense to instantiate them multiple
       times.
39 // If unsure, use this type.
40 // PluginClassName must have this constructor and a static buildPluginFactory() function to declare the
       parameters:
41 //
42 // PluginClassName(iris::IrisInstantiationContext& context) { ... initialize plugin ... }
43 // static void buildPluginFactory(iris::IrisPluginFactoryBuilder& b) { ... declare parameters ... }
44 //
45 // 2. Non-factory plugins:
46 //
47 // IRIS_NON_FACTORY_PLUGIN(PluginClassName)
48 //
49 // where PluginClassName is the class of the plugin.
50 //
51 // This declares a plugin which is automatically instantiated exactly once when the DSO is loaded.
52 // The plugin cannot have parameters and cannot be instantiated multiple times. A non-factory plugin
53 // plays the same role as the factory instance of factory plugins.
54 //
55 // PluginClassName must have this constructor:
56 //
57 // PluginClassName(iris::IrisInstantiationContext& context) { ... initialize plugin ... }
58 //
59 // Both types of plugins have identical entry points (irisInitPlugin()), and the plugin loader treats
       them the same way.
60 // After loading a plugin DSO, the plugin loader calls irisInitPlugin() which creates a single plugin
       instance.
61 // This is either a plugin factory, indicated by the fact that this instance has the functions
       plugin_getInstantiationParameterInfo()
62 // and plugin_instantiate(), or a non-factory plugin, when these plugin_*() functions are not present. In
       the latter case the
63 // plugin loader is now done. For factory-plugins the plugin loader now instantiates all desired plugins
       by calling plugin_instantiate()
64 // with the respective parameter values.
65
69 class IrisPluginFactoryBuilder : public IrisInstanceFactoryBuilder
70 {
```

```
71 private:
74     std::string plugin_name;
75
77     std::string instance_name_prefix;
78
80     std::string default_instance_name;
81
82 public:
86     IrisPluginFactoryBuilder(const std::string& name)
87         : IrisInstanceFactoryBuilder(/*parameter_prefix=*/"")
88         , plugin_name(name)
89         , instance_name_prefix("client.plugin")
90     {
91     }
92
101     void setPluginName(const std::string& name)
102     {
103         plugin_name = name;
104     }
105
110     const std::string& getPluginName() const
111     {
112         return plugin_name;
113     }
114
123     void setInstanceNamePrefix(const std::string& prefix)
124     {
125         instance_name_prefix = prefix;
126     }
127
132     const std::string& getInstanceNamePrefix() const
133     {
134         return instance_name_prefix;
135     }
136
145     void setDefaultInstanceName(const std::string& name)
146     {
147         default_instance_name = name;
148     }
149
155     const std::string& getDefaultInstanceName() const
156     {
157         if (default_instance_name.empty())
158         {
159             return getPluginName();
160         }
161         else
162         {
163             return default_instance_name;
164         }
165     }
166 };
167
168 template <class PLUGIN_CLASS>
169 class IrisPluginFactory
170 {
171 private:
173     IrisCConnection connection_interface;
174
176     IrisInstance factory_instance;
177
179     std::vector<PLUGIN_CLASS*> plugin_instances;
180
181     std::mutex plugin_instances_mutex;
182
184     IrisPluginFactoryBuilder builder;
185
187     void impl_plugin_getInstantiationParameterInfo(IrisReceivedRequest& req)
188     {
189         factory_instance.sendResponse(req.generateOkResponse(builder.getParameterInfo()));
190     }
191
193     void impl_plugin_instantiate(IrisReceivedRequest& req)
194     {
195         InstantiationResult result;
196         result.success = true; // Assume we will succeed until proven otherwise
197
198         uint64_t instance_flags = IrisInstance::DEFAULT_FLAGS;
199
200         std::string instName;
201
202         if (!req.getOptionalArg(ISTR("instName"), instName))
203         {
204             instName = builder.getDefaultInstanceName();
205             instance_flags |= IrisInstance::UNIQUIFY;
206         }
207
```

```
208         std::vector<InstantiationParameterValue> param_values;
209         req.getOptionalArg(ISTR("paramValues"), param_values);
210 
211         // Build the full parameter info list
212         const std::vector<ResourceInfo>& param_info        = builder.getParameterInfo();
213         const std::vector<ResourceInfo>& hidden_param_info = builder.getHiddenParameterInfo();
214 
215         std::vector<ResourceInfo> all_param_info;
216         all_param_info.insert(all_param_info.end(), param_info.begin(), param_info.end());
217         all_param_info.insert(all_param_info.end(), hidden_param_info.begin(), hidden_param_info.end());
218 
219         IrisInstantiationContext init_context(&connection_interface, result,
220                                               all_param_info, param_values,
221                                               builder.getInstanceNamePrefix(),
222                                               instName, instance_flags);
223 
224         // Parameters have been validated. If they all passed we can instantiate the plugin.
225 
226         if (result.success)
227         {
228             try
229             {
230                 std::lock_guard<std::mutex> lock(plugin_instances_mutex);
231 
232                 plugin_instances.push_back(new PLUGIN_CLASS(init_context));
233 
234                 if (!result.success)
235                 {
236                     // The plugin instance set an error in its constructor so destroy it.
237                     delete plugin_instances.back();
238                     plugin_instances.pop_back();
239                 }
240             }
241             catch (IrisErrorException& e)
242             {
243                 result.success = false;
244                 result.errors.resize(result.errors.size() + 1);
245 
246                 InstantiationError& error = result.errors.back();
247                 error.severity            = "error";
248                 error.code                = "error_general_error";
249                 error.message             = e.getMessage();
250             }
251             catch (...)
252             {
253                 result.success = false;
254                 result.errors.resize(result.errors.size() + 1);
255 
256                 InstantiationError& error = result.errors.back();
257                 error.severity            = "error";
258                 error.code                = "error_general_error";
259                 error.message             = "Internal error while instantiating plugin";
260             }
261         }
262 
263         factory_instance.sendResponse(req.generateOkResponse(result));
264     }
265 
266 public:
267     IrisPluginFactory(IrisC_Functions* iris_c_functions, const std::string& plugin_name)
268         : connection_interface(iris_c_functions)
269         , factory_instance(&connection_interface)
270         , builder(plugin_name)
271     {
272         PLUGIN_CLASS::buildPluginFactory(builder);
273 
274         typedef IrisPluginFactory<PLUGIN_CLASS> Self;
275 
276         factory_instance.irisRegisterFunction(this, Self, plugin_getInstantiationParameterInfo,
277                                               function_info::plugin_getInstantiationParameterInfo);
278 
279         factory_instance.irisRegisterFunction(this, Self, plugin_instantiate,
280                                               "{description:'Instantiate an instance of the " +
    builder.getPluginName() +
281                                                   " plugin',"
282                                                   "args:{"
283                                                   "  instName:{type:'String', description:'Used to
    construct the instance name for the new instance."
284                                                   " Instance name will be \""
285                                                   + builder.getInstanceNamePrefix() +
286                                                   "<instName>\"',"
287                                                   "defval:'"
288                                                   + builder.getDefaultInstanceName() +
289                                                   "', optional:true},"
290                                                   "  paramValues:{type:'Array',
    description:'Instantiation parameter values'}"
291                                                   "},"
```

```
292                                                 "retval:{type:'InstantiationResult',
    description:'Indicates success of and errors/warnings"
293                                                 " that occurred during plugin instantiation.'}}");
294
295         // Register factory instance
296         uint64_t flags = IrisInstance::DEFAULT_FLAGS
297             | IrisInstance::UNIQUIFY;
298
299         std::string factory_instName = "framework.plugin." + builder.getPluginName() + "Factory";
300         factory_instance.registerInstance(factory_instName, flags);
301         factory_instance.setProperty("componentType", "IrisPluginFactory");
302
303         IrisLogger log("IrisPluginFactory");
304     }
305
306     ~IrisPluginFactory()
307     {
308         {
309             std::lock_guard<std::mutex> lock(plugin_instances_mutex);
310
311             // Clean up plugin instances
312             typename std::vector<PLUGIN_CLASS*>::iterator it;
313             for (it = plugin_instances.begin(); it != plugin_instances.end(); ++it)
314             {
315                 delete *it;
316             }
317         }
318     }
319
320     // Unregister factory instance. Call this when unloading a plugin before simulation termination.
321     IrisErrorCode unregisterInstance()
322     {
323         return factory_instance.unregisterInstance();
324     }
325
326     // Implementation of the plugin entry point.
327     // This will initialize an IrisPluginFactory the first time it is called.
328     static int64_t initPlugin(IrisC_Functions* functions, const std::string& plugin_name)
329     {
330         static IrisPluginFactory<PLUGIN_CLASS>* factory = nullptr;
331
332         if (factory == nullptr)
333         {
334             factory = new IrisPluginFactory<PLUGIN_CLASS>(functions, plugin_name);
335             return E_ok;
336         }
337         else
338         {
339             return E_plugin_already_loaded;
340         }
341     }
342 };
343
349 #define IRIS_PLUGIN_FACTORY(PluginClassName)                                                    \
350     extern "C" IRIS_EXPORT int64_t irisInitPlugin(IrisC_Functions* functions)                   \
351     {                                                                                           \
352         return ::iris::IrisPluginFactory<PluginClassName>::initPlugin(functions, #PluginClassName); \
353     }
354
355
356 // --- Non-factory plugin support. ---
357 // Non-factory plugins are plugins which instantiate themselves directly in the entry point function.
358 // There is no factory instance. The singleton instance is the plugin rather than used to instantiate
    the plugins.
359 // They cannot receive partameters and cannot be instantiated multiple times.
360 // These are usually very simple singleton plugins.
361
369 template<class PLUGIN_CLASS>
370 class IrisNonFactoryPlugin
371 {
372 public:
373     IrisNonFactoryPlugin(IrisC_Functions* functions, const std::string& pluginName)
374         : connectionInterface(functions)
375         , instantiationContext(&connectionInterface, instantiationResult,
    std::vector<iris::ResourceInfo>(), std::vector<iris::InstantiationParameterValue>(), "client.plugin",
    pluginName, iris::IrisInstance::DEFAULT_FLAGS | iris::IrisInstance::UNIQUIFY)
376         , plugin(instantiationContext)
377     {
378     }
379
380     // Implementation of the plugin entry point.
381     // This will instantiate a new plugin.
382     static int64_t initPlugin(IrisC_Functions* functions, const std::string& pluginName)
383     {
384         new IrisNonFactoryPlugin<PLUGIN_CLASS>(functions, pluginName);
385         return E_ok;
386     }
```

```
387
388 private:
390     iris::IrisCConnection connectionInterface;
391
393     iris::IrisInstantiationContext instantiationContext;
394
396     PLUGIN_CLASS plugin;
397
399     iris::InstantiationResult instantiationResult;
400 };
401
407 #define IRIS_NON_FACTORY_PLUGIN(PluginClassName)                                                    \
408 extern "C" IRIS_EXPORT int64_t irisInitPlugin(IrisC_Functions* functions)                           \
409 {                                                                                                   \
410     return ::iris::IrisNonFactoryPlugin<PluginClassName>::initPlugin(functions, #PluginClassName);  \
411 }
412
413 NAMESPACE_IRIS_END
414
415 #endif // ARM_INCLUDE_IrisPluginFactory_h
```

9.55 IrisRegisterEventEmitter.h File Reference

Utility classes for emitting register read and register update events.

```
#include "iris/detail/IrisCommon.h"
#include "iris/detail/IrisRegisterEventEmitterBase.h"
```

Classes

- class iris::IrisRegisterReadEventEmitter< REG_T, ARGS >

  An EventEmitter class for register read events.
- class iris::IrisRegisterUpdateEventEmitter< REG_T, ARGS >

  An EventEmitter class for register update events.

9.55.1 Detailed Description

Utility classes for emitting register read and register update events.

Copyright

9.56 IrisRegisterEventEmitter.h

Go to the documentation of this file.

```
1
8 #ifndef ARM_INCLUDE_IrisRegisterEventEmitter_h
9 #define ARM_INCLUDE_IrisRegisterEventEmitter_h
10
11 #include "iris/detail/IrisCommon.h"
12 #include "iris/detail/IrisRegisterEventEmitterBase.h"
13
14 NAMESPACE_IRIS_START
15
57 template <typename REG_T, typename... ARGS>
58 class IrisRegisterReadEventEmitter : public IrisRegisterEventEmitterBase
59 {
60 public:
61     IrisRegisterReadEventEmitter()
62         : IrisRegisterEventEmitterBase(sizeof...(ARGS) + 3)
63     {
64     }
65
74     void operator()(ResourceId rscId, bool debug, REG_T value, ARGS... args)
75     {
76         // Emit event
77         emitEvent(rscId, debug, value, args...);
78
79         // Check if this event indicates a breakpoint was hit
80         if (!debug)
81         {
```

```
82             checkBreakpointHit(rscId, value, /*is_read=*/true);
83         }
84     }
85 };
86
126 template <typename REG_T, typename... ARGS>
127 class IrisRegisterUpdateEventEmitter : public IrisRegisterEventEmitterBase
128 {
129 public:
130     IrisRegisterUpdateEventEmitter()
131         : IrisRegisterEventEmitterBase(sizeof...(ARGS) + 4)
132     {
133     }
134
144     void operator()(ResourceId rscId, bool debug, REG_T old_value, REG_T new_value, ARGS... args)
145     {
146         // Emit event
147         emitEvent(rscId, debug, old_value, new_value, args...);
148
149         // Check if this event indicates a breakpoint was hit
150         if (!debug)
151         {
152             checkBreakpointHit(rscId, new_value, /*is_read=*/false);
153         }
154     }
155 };
156
157 NAMESPACE_IRIS_END
158
159 #endif // ARM_INCLUDE_IrisRegisterEventEmitter_h
```

9.57 IrisTcpClient.h File Reference

IrisTcpClient Type alias for IrisClient.
`#include "iris/IrisClient.h"`

Typedefs

- using **iris::IrisTcpClient** = IrisClient

  Alias for backward compatibility.

9.57.1 Detailed Description

IrisTcpClient Type alias for IrisClient.

Date

Copyright ARM Limited 2022 All Rights Reserved.

9.58 IrisTcpClient.h

Go to the documentation of this file.

```
1
7 #ifndef ARM_INCLUDE_IrisTcpClient_h
8 #define ARM_INCLUDE_IrisTcpClient_h
9
10 #include "iris/IrisClient.h"
11
12 NAMESPACE_IRIS_START
13
15 using IrisTcpClient = IrisClient;
16
17 NAMESPACE_IRIS_END
18
19 #endif // #ifndef ARM_INCLUDE_IrisTcpClient_h
```