IrisSupportLib

Version 1.0

Reference Guide

Chapter 1

IrisSupportLib Reference Guide

About this book

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|IrisSupport.lib. Headers are provided in the directory $IRIS_HOME/include/iris/ and the source code is provided in the directory $IRIS_HOME/↩ IrisSupportLib/.

Other information

For more information about Iris, see the *Iris User Guide*.
See the following locations for examples of Iris clients and plug-ins:

- $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.

Feedback

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

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content If you have any comments on content, send an e-mail to *errata@arm.com*. Give:

- The title *IrisSupportLib Reference Guide*.
- The number 101319_0100_15_en.
- If applicable, the relevant page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

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*.

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

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

Web Address
http://www.arm.com.

Release Information

Document History			
Issue	**Date**	**Confidentiality**	**Change**

Document History			
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.

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 uint64_t **iris::IrisInstance::DEFAULT_FLAGS** = THROW_ON_ERROR

  Default flags used if not otherwise specified.
- 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

rsc↩ Id	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

rsc↩ Id	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.
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::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, IrisRegisterEventEmitterBase &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, IrisRegisterEventEmitterBase &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 set↩ RegisterWriteEvent().

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 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.

7.3.2.10 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.

7.3.2.11 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.

7.3.2.12 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.

7.3.2.13 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.

7.3.2.14 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.

7.3.2.15 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.

7.3.2.16 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.

7.3.2.17 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.
- template<IrisErrorCode(*)(const BreakpointHitInfo &) FUNC>
  void iris::IrisInstanceBuilder::setHandleBreakpointHitDelegate ()

  Set the delegate that is called when a breakpoint is hit.
- void iris::IrisInstanceBuilder::setHandleBreakpointHitDelegate (HandleBreakpointHitDelegate delegate)

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

  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.
pc↩ SpaceId	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

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 setHandleBreakpointHitDelegate() [1/3]

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

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

Usage: Pass in a global function to call when a breakpoint is hit.

```
iris::IrisErrorCode handleBreakpointHit(const iris::BreakpointHitInfo&);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setHandleBreakpointHitDelegate<&handleBreakpointHit>();
```

Template Parameters

FUNC	Global function to call when a breakpoint is hit.

7.4.2.12 setHandleBreakpointHitDelegate() [2/3]

```
void iris::IrisInstanceBuilder::setHandleBreakpointHitDelegate (
            HandleBreakpointHitDelegate delegate )  [inline]
```

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

Usage: Pass a handle breakpoint hit delegate:

```
class MyClass
{
    ...
    iris::IrisErrorCode handleBreakpointHit(const iris::BreakpointHitInfo&);
};
MyClass myInstanceOfMyClass;
HandleBreakpointHitDelegate delegate = HandleBreakpointHitDelegate::make<MyClass,
       &MyClass::handleBreakpointHit>(myInstanceOfMyClass);
iris::IrisInstanceBuilder *builder = myIrisInstance.getBuilder();
builder->setHandleBreakpointHitDelegate(delegate);
```

Parameters

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

7.4.2.13 setHandleBreakpointHitDelegate() [3/3]

```
template<typename T , IrisErrorCode(T::*)(const BreakpointHitInfo &) METHOD>
void iris::IrisInstanceBuilder::setHandleBreakpointHitDelegate (
```

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

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

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

Template Parameters

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

Parameters

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

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.
- typedef IrisDelegate< uint64_t, const std::string &, MemoryReadResult &, uint64_t, uint64_t, std::vector< DisassemblyLine > & > iris::GetDisassemblyDelegate

  Get the disassembly of a chunk of memory.

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 (GetDisassemblyDelegate delegate)

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

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.

7.9.2.3 GetDisassemblyDelegate

```
typedef IrisDelegate<uint64_t, const std::string&, MemoryReadResult&, uint64_t, uint64_t, std::vector<DisassemblyLine>&> iris::GetDisassemblyDelegate
```

Get the disassembly of a chunk of memory.

```
IrisErrorCode getDisassembly(uint64_t address, const std::string &mode, MemoryReadResult &memoryReadData,
                             uint64_t count, uint64_t maxAddr, std::vector<DisassemblyLine>
      &disassemblyLineOut)
```

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

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 (
            GetDisassemblyDelegate 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**
- bool **isReadAccess**

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

- IrisInstanceBreakpoint.h

8.4 iris::IrisInstanceBuilder::EventSourceBuilder Class Reference

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

Public Member Functions

- 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 & 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.

8.4.1 Detailed Description

Used to set metadata on an EventSource.

8.4.2 Member Function Documentation

8.4.2.1 addEnumElement()

```
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.

8.4.2.2 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.

8.4.2.3 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.

8.4.2.4 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.

8.4.2.5 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.

8.4.2.6 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.

8.4.2.7 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.

8.4.2.8 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.

8.4.2.9 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.

8.4.2.10 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.

8.4.2.11 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

8.5 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**

8.5.1 Detailed Description

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

- IrisInstanceEvent.h

8.6 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.
- 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, const EventSourceInfo *srcInfo, 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**

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

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

  The event stream id.
- IrisU64JsonWriter::Object **fieldObj**
- IrisRequest * **internal_req**
- IrisInstance * irisInstance

  — basic members —
- 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.
- const EventSourceInfo * **srcInfo**

  The event source info.
- uint64_t **startVal**

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

  Synchronous callback behavior.

8.6.1 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.

8.6.2 Member Function Documentation

8.6.2.1 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.

8.6.2.2 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.

8.6.2.3 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.

8.6.2.4 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.

8.6.2.5 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.

8.6.2.6 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.

8.6.2.7 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.

8.6.2.8 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.

8.6.2.9 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.

8.6.2.10 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.

8.6.2.11 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.

8.6.2.12 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.

8.6.2.13 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.

8.6.2.14 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.

8.6.2.15 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.

8.6.2.16 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.

8.6.2.17 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.

8.6.2.18 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.

8.6.2.19 getCountVal()

`uint64_t iris::EventStream::getCountVal ( ) const [inline]`
Get the current value of the counter.

Returns

The current value of the event counter.

8.6.2.20 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.

8.6.2.21 getEsId()

`EventStreamId iris::EventStream::getEsId ( ) const [inline]`
Get the Id of this event stream.

Returns

The esId for this event stream.

8.6.2.22 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.

8.6.2.23 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

8.6.2.24 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.

8.6.2.25 isCounter()

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

Is this event stream a counter?

Returns

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

8.6.2.26 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.

8.6.2.27 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`.

8.6.2.28 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`.

8.6.2.29 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'.

8.6.2.30 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.

8.6.2.31 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).
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.

8.6.2.32 setProperties()

```
void iris::EventStream::setProperties (
            IrisInstance * irisInstance,
```

```
            const EventSourceInfo * srcInfo,
            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.
srcInfo	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.

8.6.2.33 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

8.6.2.34 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.

8.6.3 Member Data Documentation

8.6.3.1 counter

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

— members for a counter —
Is a counter?

8.6.3.2 irisInstance

`IrisInstance* iris::EventStream::irisInstance [protected]`
— basic members —
The Iris instance that created this event.

8.6.3.3 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

8.7 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 & 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)

  Obsolete alias for setDescription(). Do not use.
- 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** {}

8.7.1 Detailed Description

Used to set metadata on a register field resource.

8.7.2 Member Function Documentation

8.7.2.1 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.

8.7.2.2 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

8.7.2.3 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

8.7.2.4 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.

8.7.2.5 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.

8.7.2.6 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.

8.7.2.7 parent()

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

Get the RegisterBuilder for the parent register.

Returns

The RegisterBuilder object for the parent register.

8.7.2.8 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.

8.7.2.9 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.

8.7.2.10 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.

8.7.2.11 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.

8.7.2.12 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.

8.7.2.13 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.

8.7.2.14 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.

8.7.2.15 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.

8.7.2.16 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.

8.7.2.17 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.

8.7.2.18 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.

8.7.2.19 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.

8.7.2.20 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.

8.7.2.21 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.

8.7.2.22 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.

8.7.2.23 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.

8.7.2.24 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.

8.7.2.25 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.

8.7.2.26 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.

8.7.2.27 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.

8.7.2.28 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.

8.7.2.29 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.

8.7.2.30 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.

8.7.2.31 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.

8.7.2.32 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.

8.7.2.33 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.

8.7.2.34 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.

8.7.2.35 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

8.8 iris::IrisCConnection Class Reference

Provide an IrisConnectionInterface which loads an IrisC library.
`#include <IrisCConnection.h>`
Inherits IrisConnectionInterface.

Public Member Functions

- virtual IrisInterface * **getIrisInterface** () IRIS_OVERRIDE

  Get the IrisInterface for this connection. See also IrisConnectionInterface::getIrisInterface().
- **IrisCConnection** (IrisC_Functions *functions)
- virtual IrisErrorCode **processAsyncMessages** (bool waitForAMessage) IRIS_OVERRIDE

  Process asynchronous messages for the calling thread. See also IrisConnectionInterface::processAsyncMessages().
- virtual uint64_t **registerIrisInterfaceChannel** (IrisInterface *iris_interface) IRIS_OVERRIDE

  Register a communication channel. See also IrisConnectionInterface::registerIrisInterfaceChannel().
- virtual void **unregisterIrisInterfaceChannel** (uint64_t channelId) IRIS_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.

8.8.1 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

8.9 iris::IrisClient Class Reference

Inherits IrisInterface, impl::IrisProcessEventsInterface, and IrisConnectionInterface.

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 connectSocketFd (SocketFd socketfd, unsigned timeoutInMs=1000)
- IrisErrorCode disconnect ()
- bool disconnectAndWaitForChildToExit (double timeoutInMs=5000, double timeoutInMsAfterSigInt=5000, double timeoutInMsAfterSigKill=5000)
- pid_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=std::string(), const std::string &connectionSpec=std::string())

  Client constructor.
- 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)
- uint64_t **registerChannel** (IrisC_CommunicationChannel ∗channel, const ::std::string &path)
- virtual uint64_t **registerIrisInterfaceChannel** (IrisInterface ∗iris_interface) override
- void setInstanceName (const std::string &instName)
- void **setIrisMessageLogLevel** (unsigned level)

  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 (pid_t pid, int ∗status, int options, double timeoutInMs)
- virtual ∼**IrisClient** ()

  Destructor.

Public Attributes

- const std::string connectionHelpStr

  Connection help string.

8.9.1 Constructor & Destructor Documentation

8.9.1.1 IrisClient()

```
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.

8.9.2 Member Function Documentation

8.9.2.1 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.

8.9.2.2 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.

8.9.2.3 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.

8.9.2.4 disconnect()

```
IrisErrorCode iris::IrisClient::disconnect ( )  [inline]
```

Disconnect from server. Close socket. (Only for mode IRIS_TCP_CLIENT.)

8.9.2.5 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 timeoutInMsAfterSigInt until the child exits. If the child did not exit by then, send a SIGKILL and wait for timeoutInMsAfterSigKill 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.

8.9.2.6 getIrisInstance()

```
IrisInstance & iris::IrisClient::getIrisInstance ( )  [inline]
```

Get contained IrisInstance. This can be used as a generic client instance to call Iris functions.

8.9.2.7 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. (TODO safer memory management of this object) -return Nothing.

8.9.2.8 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

8.9.2.9 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.

8.9.2.10 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().

8.9.2.11 setSleepOnDestructionMs()

```
void iris::IrisClient::setSleepOnDestructionMs (
            uint64_t sleepOnDestructionMs_ )  [inline]
```

Sleep a short time on destruction to de-interleave output by different processes. This has not functional impacto or purpose. It just beautifies the output on stdout.

8.9.2.12 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.

8.9.2.13 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.

8.9.2.14 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().

8.9.2.15 waitpidWithTimeout()

```
bool iris::IrisClient::waitpidWithTimeout (
            pid_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.

8.9.3 Member Data Documentation

8.9.3.1 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"
```

Connection help string.

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

- IrisClient.h

8.10 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=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
- const std::vector< std::string > & **getNonOptionArguments** () const

  Get non-option arguments.
- 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_)

  Constructor.
- 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 IrisErrorException &e) const
- 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 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 unsetValue (const std::string &longOption)

Static Public Member Functions

- static int defaultMessageFunc (const std::string &message, int error, bool exit)

8.10.1 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:

8.10.2 Member Function Documentation

8.10.2.1 addOption()

```
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 'formalArgumentName' 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.

8.10.2.2 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.

8.10.2.3 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.

8.10.2.4 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.)

8.10.2.5 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.)

8.10.2.6 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.)

8.10.2.7 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.

8.10.2.8 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.)

8.10.2.9 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.)

8.10.2.10 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.

8.10.2.11 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.

8.10.2.12 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.

8.10.2.13 printErrorAndExit() [1/3]

```
int iris::IrisCommandLineParser::printErrorAndExit (
            const IrisErrorException & e ) const  [inline]
```

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.

8.10.2.14 printErrorAndExit() [2/3]

```
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.

8.10.2.15 printErrorAndExit() [3/3]

```
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.

8.10.2.16 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().

8.10.2.17 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 parserCommandLine() for errors raised by parseCommandLine(). For errors in the get*() functions the return value is ignored.

8.10.2.18 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.

8.10.2.19 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

8.11 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.

8.11.1 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);
```

8.11.2 Member Function Documentation

8.11.2.1 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

8.12 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.

8.12.1 Detailed Description

Class to register Iris event streams for an event.

8.12.2 Member Function Documentation

8.12.2.1 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.

8.12.2.2 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.

8.12.2.3 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.

8.12.2.4 emitEventEnd()

```
void iris::IrisEventRegistry::emitEventEnd ( ) const
```

Emit the callback.
This also checks the ranges and maintains the counter.

8.12.2.5 empty()

```
bool iris::IrisEventRegistry::empty ( ) const  [inline]
```

Return true if no event streams are registered.

Returns

`true` if no event streams are registered.

8.12.2.6 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.

8.12.2.7 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).

8.12.2.8 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.

8.12.2.9 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

8.13 iris::IrisEventStream Class Reference

Event stream class for Iris-specific events.
`#include <IrisInstanceEvent.h>`
Inherits iris::EventStream.

Public Member Functions

- virtual IrisErrorCode disable () IRIS_OVERRIDE

  Disable this event stream.
- virtual IrisErrorCode enable () IRIS_OVERRIDE

  Enable this event stream.
- **IrisEventStream** (IrisEventRegistry *registry_)

Additional Inherited Members

8.13.1 Detailed Description

Event stream class for Iris-specific events.

8.13.2 Member Function Documentation

8.13.2.1 disable()

```
virtual IrisErrorCode iris::IrisEventStream::disable ( )  [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.

8.13.2.2 enable()

```
virtual IrisErrorCode iris::IrisEventStream::enable ( )  [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

8.14 iris::IrisGlobalInstance Class Reference

Inherits IrisInterface, and IrisConnectionInterface.

Public Member Functions

- 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 &connection_info="")
- virtual uint64_t registerIrisInterfaceChannel (IrisInterface *iris_interface) override
- virtual void **setIrisProxyInterface** (IrisProxyInterface *irisProxyInterface_) override

  Set proxy interface.
- void **setLogLevel** (unsigned level)
- void **unregisterChannel** (uint64_t channelId)

  Unregister a channel.
- virtual void unregisterIrisInterfaceChannel (uint64_t channelId) override
- ~**IrisGlobalInstance** ()

  Destructor.

8.14.1 Member Function Documentation

8.14.1.1 getIrisInstance()

```
IrisInstance & iris::IrisGlobalInstance::getIrisInstance ( )  [inline]
```
Get contained IrisInstance. This can be used as a generic client instance to call Iris functions.

8.14.1.2 registerChannel()

```
uint64_t iris::IrisGlobalInstance::registerChannel (
            IrisC_CommunicationChannel * channel,
            const std::string & connection_info = "" )
```
Register a channel. Returns an associated channel id.

8.14.1.3 registerIrisInterfaceChannel()

```
virtual uint64_t iris::IrisGlobalInstance::registerIrisInterfaceChannel (
            IrisInterface * iris_interface )  [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.

8.14.1.4 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

8.15 iris::IrisInstance Class Reference

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 disableEvent (const std::string &eventSpec)

  Disable all matching event callback(s).
- void enableEvent (const std::string &eventSpec, std::function< void()> callback)

  Enable event callback(s).
- void enableEvent (const std::string &eventSpec, std::function< void(const EventStreamInfo &eventStream↩ Info, IrisReceivedRequest &request)> callback)

  Enable event callback(s).
- 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.
- 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.
- 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 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 resource infos.
- 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 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.
- 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 instaneRegistry_registerInstance().
- void setPendingSyncStepResponse (RequestId requestId, EventBufferId evBufId)

  Set pending response to a step_syncStep() call.
- template<class T >
  void setProperty (const std::string &propertyName, const T &propertyValue)

  Set/add instance property.
- 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 uint64_t **DEFAULT_FLAGS** = THROW_ON_ERROR

  Default flags used if not otherwise specified.
- 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

- InstanceInfo **thisInstanceInfo** {}

  InstanceInfo of this instance.

8.15.1 Member Typedef Documentation

8.15.1.1 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.

8.15.2 Constructor & Destructor Documentation

8.15.2.1 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."
flags	A bitwise OR of Instance Flags. Client instances should usually set the flag iris::IrisInstance::UNIQUIFY.

8.15.2.2 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.

8.15.3 Member Function Documentation

8.15.3.1 addCallback_IRIS_INSTANCE_REGISTRY_CHANGED()

```
void iris::IrisInstance::addCallback_IRIS_INSTANCE_REGISTRY_CHANGED (
            EventCallbackFunction f )
```

Add callback function for IRIS_INSTANCE_REGISTRY_CHANGED.

8.15.3.2 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");

8.15.3.3 enableEvent() [1/2]

```
void iris::IrisInstance::enableEvent (
            const std::string & eventSpec,
            std::function< void()> callback )
```

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 ... });

8.15.3.4 enableEvent() [2/2]

```
void iris::IrisInstance::enableEvent (
            const std::string & eventSpec,
            std::function< void(const EventStreamInfo &eventStreamInfo, IrisReceivedRequest
&request)> callback )
```

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.

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 ... }); });
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"; });
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.

8.15.3.5 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.

8.15.3.6 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 defaultInstId.
- 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.

8.15.3.7 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.

8.15.3.8 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.

8.15.3.9 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.

8.15.3.10 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.

8.15.3.11 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.

8.15.3.12 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.

8.15.3.13 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.

8.15.3.14 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.

8.15.3.15 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.

8.15.3.16 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.

8.15.3.17 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.

8.15.3.18 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.

8.15.3.19 getPropertyMap()

```
const PropertyMap & iris::IrisInstance::getPropertyMap ( ) const  [inline]
```

Get property map.
This can be used to lookup properties: getWithDefault(my_instance->getPropertyMap(), "myStringProperty", "").getAsString();

8.15.3.20 getRemoteIrisInterface()

```
IrisInterface * iris::IrisInstance::getRemoteIrisInterface ( )  [inline]
```

Get the remote Iris interface.

Returns

Returns the IrisInterface that this instance sends requests and responses to.

8.15.3.21 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.

8.15.3.22 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()`.

8.15.3.23 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(...);
```

8.15.3.24 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)
{
    ...
}
```

8.15.3.25 isRegistered()

```
bool iris::IrisInstance::isRegistered ( ) const  [inline]
```

Return true iff we are registered as an instance (= we have a valid instance id).

8.15.3.26 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.

8.15.3.27 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.

8.15.3.28 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.

8.15.3.29 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.

8.15.3.30 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.

8.15.3.31 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.

8.15.3.32 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.

Parameters

dlgInstanceTypeStr	The name of the delegate type. This is only used for logging purposes.

8.15.3.33 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.

8.15.3.34 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.

8.15.3.35 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, implFunctionName, 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.

8.15.3.36 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.

8.15.3.37 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::Elf↩ Dwarf::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)

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 well-known registers easy (e.g. PC, instruction counter). This intentionally does not handle the generic case (string registers, wide registers) to keep the usage simple. Use resource_read() to read any register which does not fit this function.
The resource meta-information is cached in this instance, but the value is not. The cache can be cleared with clearCachedMetaInfo().

Returns

Resource value.

8.15.3.38 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.

8.15.3.39 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 an empty string.
See resourceRead() for semantics of resourceSpec, errors and limitations.

8.15.3.40 resourceWrite()

```
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.

8.15.3.41 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.

8.15.3.42 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.

8.15.3.43 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.

8.15.3.44 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.

8.15.3.45 setCallback_IRIS_SHUTDOWN_LEAVE()

```
void iris::IrisInstance::setCallback_IRIS_SHUTDOWN_LEAVE (
            EventCallbackFunction f )
```

Set callback function for IRIS_SHUTDOWN_LEAVE.

8.15.3.46 setCallback_IRIS_SIMULATION_TIME_EVENT()

```
void iris::IrisInstance::setCallback_IRIS_SIMULATION_TIME_EVENT (
            EventCallbackFunction f )
```

Set callback function for IRIS_SIMULATION_TIME_EVENT.

8.15.3.47 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.

8.15.3.48 setPendingSyncStepResponse()

```
void iris::IrisInstance::setPendingSyncStepResponse (
            RequestId requestId,
            EventBufferId evBufId )
```

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.

8.15.3.49 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.
propertyValue	Value of the property.

8.15.3.50 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().

8.15.3.51 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.

8.15.3.52 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.

8.15.3.53 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.

8.15.3.54 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.

8.15.3.55 simulationTimeStop()

```
void iris::IrisInstance::simulationTimeStop ( )
```

Stop simulation time and wait until simulation time stopped.
This function throws Iris errors.

8.15.3.56 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.

8.15.3.57 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.

8.15.3.58 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

8.16 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 handleBreakpointHit (const BreakpointHitInfo &bptHitInfo)

  Handle breakpoint hit.
- **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 setHandleBreakpointHitDelegate (HandleBreakpointHitDelegate delegate)

  Set a delegate for handling breakpoint hit in this instance.

8.16.1 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.

8.16.2 Member Function Documentation

8.16.2.1 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.
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.

8.16.2.2 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.

8.16.2.3 getBreakpointInfo()

```
const BreakpointInfo * iris::IrisInstanceBreakpoint::getBreakpointInfo (
            BreakpointId bptId ) const
```

Get BreakpointInfo for a breakpoint id.

Parameters

bptId	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.

8.16.2.4 handleBreakpointHit()

```
void iris::IrisInstanceBreakpoint::handleBreakpointHit (
            const BreakpointHitInfo & bptHitInfo )
```

Handle breakpoint hit.

Parameters

bptHitInfo	The information of the breakpoint that is hit. Calls a delegate method in the model.

8.16.2.5 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.
pcSpaceId	Memory space Id for the memory space that the PC address corresponds to.

8.16.2.6 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.

8.16.2.7 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.
pcSpaceId	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.

8.16.2.8 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.

8.16.2.9 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.

8.16.2.10 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.

8.16.2.11 setHandleBreakpointHitDelegate()

```
void iris::IrisInstanceBreakpoint::setHandleBreakpointHitDelegate (
            HandleBreakpointHitDelegate delegate )
```

Set a delegate for handling breakpoint hit in this instance.

Parameters

delegate	A HandleBreakpointHitDelegate to call when a breakpoint is hit.

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

- IrisInstanceBreakpoint.h

8.17 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.
- 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 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 subRscId↩ Start=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 ()
- 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 pcSpace↩ Id, 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 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.
- template<IrisErrorCode(*)(const BreakpointHitInfo &) FUNC>
  void setHandleBreakpointHitDelegate ()

  Set the delegate that is called when a breakpoint is hit.
- void setHandleBreakpointHitDelegate (HandleBreakpointHitDelegate delegate)

  Set the delegate that is called when a breakpoint is hit.
- template<typename T , IrisErrorCode(T::*)(const BreakpointHitInfo &) METHOD>
  void setHandleBreakpointHitDelegate (T *instance)

  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** (GetDisassemblyDelegate delegate)

  Set the delegate to get the disassembly of a chunk of memory.
- template<typename T , IrisErrorCode(T::*)(uint64_t, const std::string &, MemoryReadResult &, uint64_t, uint64_t, std::vector< DisassemblyLine > &) METHOD>
  void **setGetDisassemblyDelegate** (T *instance)
- template<IrisErrorCode(*)(uint64_t, const std::string &, MemoryReadResult &, uint64_t, uint64_t, std::vector< DisassemblyLine > &) FUNC>
  void **setGetDisassemblyDelegate** ()
- 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=DebuggableStateGetAcknow

  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.

8.17.1 Detailed Description

Builder interface to populate an IrisInstance with registers, memory etc.
See DummyComponent.h for a working example.

8.17.2 Constructor & Destructor Documentation

8.17.2.1 IrisInstanceBuilder()

```
iris::IrisInstanceBuilder::IrisInstanceBuilder (
            IrisInstance * iris_instance )
```

Construct an IrisInstanceBuilder for an Iris instance.

Parameters

iris_instance	The instance to build.

8.17.3 Member Function Documentation

8.17.3.1 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.

8.17.3.2 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.

8.17.3.3 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.

8.17.3.4 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.

8.17.3.5 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.

8.17.3.6 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.

8.17.3.7 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.

8.17.3.8 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.

8.17.3.9 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.

8.17.3.10 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.

8.17.3.11 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.

8.17.3.12 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.

8.17.3.13 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.

8.17.3.14 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.

8.17.3.15 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.

8.17.3.16 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.

8.17.3.17 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.

8.17.3.18 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.

8.17.3.19 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.

8.17.3.20 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.

8.17.3.21 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.

8.17.3.22 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

8.18 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.

8.18.1 Detailed Description

Checkpoint add-on for IrisInstance.

8.18.2 Member Function Documentation

8.18.2.1 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.

8.18.2.2 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.

8.18.2.3 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

8.19 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.

8.19.1 Detailed Description

Debuggable-state add-on for IrisInstance.

8.19.2 Member Function Documentation

8.19.2.1 attachTo()

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

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.19.2.2 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.

8.19.2.3 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

8.20 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 (GetDisassemblyDelegate delegate)

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

8.20.1 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

8.21 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.
- 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.
- bool hasEventSource (const std::string &eventName)

  Check if event source already exists.
- IrisInstanceEvent (IrisInstance *irisInstance=nullptr)

  Construct an IrisInstanceEvent add-on.
- bool isValidEvBufId (EventBufferId evBufId) const

  Check whether event buffer id is valid.
- void setDefaultEsCreateDelegate (EventStreamCreateDelegate delegate)

  Set the default delegate for creating EventStreams for the attached instance.

Friends

- struct **EventBuffer**

8.21.1 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.

8.21.2 Constructor & Destructor Documentation

8.21.2.1 IrisInstanceEvent()

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

Construct an IrisInstanceEvent add-on.

Parameters

irisInstance	The IrisInstance to which to attach this add-on.

8.21.3 Member Function Documentation

8.21.3.1 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.

8.21.3.2 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().

8.21.3.3 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.

8.21.3.4 deleteEventSource()

```
void iris::IrisInstanceEvent::deleteEventSource (
            const std::string & eventName )
```

Delete metadata for an event source.

Parameters

eventName	The name of the event source.

8.21.3.5 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().

8.21.3.6 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.

8.21.3.7 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().
request↩ Id	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.

8.21.3.8 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.

8.21.3.9 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.

8.21.3.10 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

8.22 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)

  Add a new boolean parameter.
- IrisParameterBuilder addHiddenBooleanParameter (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.

8.22.1 Detailed Description

A builder class to construct instantiation parameter metadata.

8.22.2 Constructor & Destructor Documentation

8.22.2.1 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.

8.22.3 Member Function Documentation

8.22.3.1 addBooleanParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addBooleanParameter (
            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.

8.22.3.2 addHiddenBooleanParameter()

```
IrisParameterBuilder iris::IrisInstanceFactoryBuilder::addHiddenBooleanParameter (
            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.

8.22.3.3 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.

8.22.3.4 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.

8.22.3.5 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.

8.22.3.6 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.

8.22.3.7 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.

8.22.3.8 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

8.23 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.

8.23.1 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.

8.23.2 Constructor & Destructor Documentation

8.23.2.1 IrisInstanceImage()

```
iris::IrisInstanceImage::IrisInstanceImage (
            IrisInstance * irisInstance = 0 )
```

Construct a new IrisInstanceImage.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.23.3 Member Function Documentation

8.23.3.1 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.

8.23.3.2 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.

8.23.3.3 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.

8.23.3.4 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

8.24 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().

8.24.1 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.

8.24.2 Constructor & Destructor Documentation

8.24.2.1 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.

8.24.3 Member Function Documentation

8.24.3.1 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.

8.24.3.2 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

8.25 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.

8.25.1 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

8.25.2 Constructor & Destructor Documentation

8.25.2.1 IrisInstanceMemory()

```
iris::IrisInstanceMemory::IrisInstanceMemory (
            IrisInstance * irisInstance = 0 )
```

Construct an IrisInstanceMemory.
Optionally attaches to an IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.25.3 Member Function Documentation

8.25.3.1 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.
out↩ SpaceId	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.

8.25.3.2 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.

8.25.3.3 attachTo()

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

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.25.3.4 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.

8.25.3.5 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.

8.25.3.6 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.

8.25.3.7 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

8.26 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.

8.26.1 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.

8.26.2 Constructor & Destructor Documentation

8.26.2.1 IrisInstancePerInstanceExecution()

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

Construct an IrisInstancePerInstanceExecution add-on.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.26.3 Member Function Documentation

8.26.3.1 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.

8.26.3.2 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.

8.26.3.3 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

8.27 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)

  Calculate hierarchicalName and hierarchicalCName for all RegisterInfos.
- static void makeNamesHierarchical (std::vector< ResourceInfo > &resourceInfos)

  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)

8.27.1 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.

8.27.2 Constructor & Destructor Documentation

8.27.2.1 IrisInstanceResource()

```
iris::IrisInstanceResource::IrisInstanceResource (
            IrisInstance * irisInstance = 0 )
```

Construct an IrisInstanceResource.
Optionally attaches to an IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.27.3 Member Function Documentation

8.27.3.1 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.
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.

8.27.3.2 attachTo()

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

Attach this IrisInstance add-on to a specific IrisInstance.

Parameters

irisInstance	The IrisInstance to attach to.

8.27.3.3 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*.

8.27.3.4 calcHierarchicalNames()

```
static void iris::IrisInstanceResource::calcHierarchicalNames (
            std::vector< ResourceInfo > & resourceInfos ) [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.

8.27.3.5 getResourceInfo()

```
ResourceInfoAndAccess * iris::IrisInstanceResource::getResourceInfo (
            ResourceId rscId )
```

Get the resource info for a resource that was already added.

Parameters

rscId	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.

8.27.3.6 makeNamesHierarchical()

```
static void iris::IrisInstanceResource::makeNamesHierarchical (
            std::vector< ResourceInfo > & resourceInfos ) [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.

8.27.3.7 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

8.27.3.8 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

8.28 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)

8.28.1 Member Function Documentation

8.28.1.1 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.

8.28.1.2 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.

8.28.1.3 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.

8.28.1.4 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

8.29 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)

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.

8.29.1 Detailed Description

An IrisInstance add-on that adds simulation functions for the SimulationEngine instance.

8.29.2 Constructor & Destructor Documentation

8.29.2.1 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.

8.29.3 Member Function Documentation

8.29.3.1 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.

8.29.3.2 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().

8.29.3.3 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.

8.29.3.4 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".

8.29.3.5 notifySimPhase()

```
void iris::IrisInstanceSimulation::notifySimPhase (
            uint64_t time,
            IrisSimulationPhase phase )
```

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.

8.29.3.6 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.

8.29.3.7 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.

8.29.3.8 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.

8.29.3.9 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.

8.29.3.10 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.

8.29.3.11 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.

8.29.3.12 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.

8.29.3.13 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.

8.29.3.14 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.

8.29.3.15 setLogLevel()

```
void iris::IrisInstanceSimulation::setLogLevel (
            unsigned logLevel_ )  [inline]
```

Set log level (0-1).
Set log level (0-1).

8.29.3.16 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.

8.29.3.17 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.

8.29.3.18 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.

8.29.3.19 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.

8.29.3.20 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.

8.29.3.21 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.

8.29.3.22 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.

8.29.3.23 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.

8.29.3.24 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.

Template Parameters

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

8.30 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.

8.30.1 Detailed Description

Simulation time add-on for IrisInstance.

8.30.2 Constructor & Destructor Documentation

8.30.2.1 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.

8.30.3 Member Function Documentation

8.30.3.1 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.

8.30.3.2 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 IrisIntanceEvent has been attached to iris_instance (IrisInstanceEvent::attachTo). This will ensure that event functions have been registered on simulation time iris instance.

8.30.3.3 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.

8.30.3.4 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.

8.30.3.5 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.

8.30.3.6 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.

8.30.3.7 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.

8.30.3.8 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.

8.30.3.9 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.

8.30.3.10 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.

8.30.3.11 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.

8.30.3.12 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.

8.30.3.13 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

8.31 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.

8.31.1 Detailed Description

Step add-on for IrisInstance.
This is used by instances to support stepping functionality.
This class implements all Iris step*() functions.

8.31.2 Constructor & Destructor Documentation

8.31.2.1 IrisInstanceStep()

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

Construct an IrisInstanceStep add-on.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.31.3 Member Function Documentation

8.31.3.1 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.

8.31.3.2 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.

8.31.3.3 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.

8.31.3.4 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

8.32 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.

8.32.1 Detailed Description

Table add-on for IrisInstance.
This is used by instances to support table functionality.

8.32.2 Constructor & Destructor Documentation

8.32.2.1 IrisInstanceTable()

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

Construct an IrisInstanceTable add-on.

Parameters

irisInstance	The IrisInstance to attach this add-on to.

8.32.3 Member Function Documentation

8.32.3.1 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.

8.32.3.2 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.

8.32.3.3 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.

8.32.3.4 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

8.33 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.

8.33.1 Detailed Description

Provides context when instantiating an Iris instance from a factory.

8.33.2 Member Function Documentation

8.33.2.1 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.

8.33.2.2 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.

8.33.2.3 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().

8.33.2.4 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().

8.33.2.5 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.

8.33.2.6 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.

8.33.2.7 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.

8.33.2.8 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().

8.33.2.9 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.

8.33.2.10 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.

8.33.2.11 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.

8.33.2.12 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.

8.33.2.13 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.

8.33.2.14 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.

8.33.2.15 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

8.34 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)

8.34.1 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

8.35 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.

8.35.1 Detailed Description

Helper class to construct instantiation parameters.

8.35.2 Constructor & Destructor Documentation

8.35.2.1 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.

8.35.3 Member Function Documentation

8.35.3.1 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.
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.

8.35.3.2 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.

8.35.3.3 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.

8.35.3.4 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.

8.35.3.5 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.

8.35.3.6 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.

8.35.3.7 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.

8.35.3.8 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.

8.35.3.9 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.

8.35.3.10 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.

8.35.3.11 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.

8.35.3.12 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.

8.35.3.13 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.

8.35.3.14 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.

8.35.3.15 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.

8.35.3.16 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.

8.35.3.17 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.

8.35.3.18 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.

8.35.3.19 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.

8.35.3.20 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.

8.35.3.21 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.

8.35.3.22 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.

8.35.3.23 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.

8.35.3.24 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.

8.35.3.25 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.

8.35.3.26 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.

8.35.3.27 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.

8.35.3.28 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.

8.35.3.29 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.

8.35.3.30 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.

8.35.3.31 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.

8.35.3.32 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.

8.35.3.33 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.

8.35.3.34 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.

8.35.3.35 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

8.36 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

8.37 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.

8.37.1 Detailed Description

Set meta data for instantiating a plug-in instance.

8.37.2 Constructor & Destructor Documentation

8.37.2.1 IrisPluginFactoryBuilder()

```
iris::IrisPluginFactoryBuilder::IrisPluginFactoryBuilder (
            const std::string & name )  [inline]
```

Parameters

name	The name of the plug-in to build.

8.37.3 Member Function Documentation

8.37.3.1 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.

8.37.3.2 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.

8.37.3.3 getPluginName()

```
const std::string & iris::IrisPluginFactoryBuilder::getPluginName ( ) const  [inline]
```

Get the plug-in name.

Returns

The name of the plug-in.

8.37.3.4 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.

8.37.3.5 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.

8.37.3.6 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

8.38 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.

8.38.1 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;
 }
```

8.38.2 Member Function Documentation

8.38.2.1 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

8.39 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.

8.39.1 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);
 }
```

8.39.2 Member Function Documentation

8.39.2.1 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

8.40 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)

8.40.1 Detailed Description

Provides context to a reset delegate call.

8.40.2 Member Function Documentation

8.40.2.1 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

8.41 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 & 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.

8.41.1 Detailed Description

Used to set metadata for a memory space.

8.41.2 Member Function Documentation

8.41.2.1 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.

8.41.2.2 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.

8.41.2.3 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.

8.41.2.4 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.

8.41.2.5 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.

8.41.2.6 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.

8.41.2.7 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.

8.41.2.8 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.

8.41.2.9 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.

8.41.2.10 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.

8.41.2.11 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.

8.41.2.12 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.

8.41.2.13 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.

8.41.2.14 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.

8.41.2.15 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.

8.41.2.16 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.

8.41.2.17 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.

8.41.2.18 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.

8.41.2.19 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

8.42 iris::IrisCommandLineParser::Option Struct Reference

Option container.
`#include <IrisCommandLineParser.h>`

Public Member Functions

- Option & setList (char sep=',')

Friends

- class **IrisCommandLineParser**

8.42.1 Detailed Description

Option container.

8.42.2 Member Function Documentation

8.42.2.1 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

8.43 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)

  Obsolete alias for setDescription(). Do not use.
- 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.

8.43.1 Detailed Description

Used to set metadata on a parameter.

8.43.2 Member Function Documentation

8.43.2.1 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.

8.43.2.2 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.

8.43.2.3 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.

8.43.2.4 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.

8.43.2.5 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.

8.43.2.6 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.

8.43.2.7 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.

8.43.2.8 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.

8.43.2.9 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.

8.43.2.10 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.

8.43.2.11 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.

8.43.2.12 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.

8.43.2.13 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.

8.43.2.14 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.

8.43.2.15 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.

8.43.2.16 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.

8.43.2.17 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.

8.43.2.18 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.

8.43.2.19 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.

8.43.2.20 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.

8.43.2.21 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.

8.43.2.22 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.

8.43.2.23 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.

8.43.2.24 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.

8.43.2.25 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.

8.43.2.26 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.

8.43.2.27 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.

8.43.2.28 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.

8.43.2.29 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.

8.43.2.30 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.

8.43.2.31 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.

8.43.2.32 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.

8.43.2.33 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

8.44 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** {}

8.44.1 Detailed Description

Contains information for a single proxy EventSource.
The documentation for this struct was generated from the following file:

- IrisInstanceEvent.h

8.45 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 & 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)

  Obsolete alias for setDescription(). Do not use.
- 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.

8.45.1 Detailed Description

Used to set metadata on a register resource.

8.45.2 Member Function Documentation

8.45.2.1 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.

8.45.2.2 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.

8.45.2.3 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.

8.45.2.4 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.

8.45.2.5 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.

8.45.2.6 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.

8.45.2.7 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.

8.45.2.8 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.

8.45.2.9 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.

8.45.2.10 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.

8.45.2.11 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.

8.45.2.12 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.

8.45.2.13 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.

8.45.2.14 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.

8.45.2.15 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.

8.45.2.16 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

parentRscId	The rscId of the parent register.

Returns

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

8.45.2.17 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.

8.45.2.18 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.

8.45.2.19 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.

8.45.2.20 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.

8.45.2.21 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.

8.45.2.22 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.

8.45.2.23 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.

8.45.2.24 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.

8.45.2.25 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.

8.45.2.26 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.

8.45.2.27 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.

8.45.2.28 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.

8.45.2.29 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.

8.45.2.30 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.

8.45.2.31 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.

8.45.2.32 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.

8.45.2.33 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.

8.45.2.34 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

8.46 iris::IrisInstanceResource::ResourceInfoAndAccess Struct Reference

Entry in 'resourceInfos'.
`#include <IrisInstanceResource.h>`

Public Attributes

- ResourceReadDelegate **readDelegate**
- ResourceInfo **resourceInfo**
- ResourceWriteDelegate **writeDelegate**

8.46.1 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

8.47 iris::ResourceWriteValue Struct Reference

`#include <IrisInstanceResource.h>`

Public Attributes

- const uint64_t * **data** {}
- const std::string * **str** {}

  Non-null for non-string resources.

8.47.1 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

8.48 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)

8.48.1 Detailed Description

semihosting_apis IrisInstanceBuilder semihosting APIs
Manage semihosting functionality

8.48.2 Member Function Documentation

8.48.2.1 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.

8.48.2.2 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

8.49 iris::IrisInstanceMemory::SpaceInfoAndAccess Struct Reference

Entry in 'spaceInfos'.
`#include <IrisInstanceMemory.h>`

Public Attributes

- MemoryReadDelegate **readDelegate**
- MemoryGetSidebandInfoDelegate **sidebandDelegate**
- MemorySpaceInfo **spaceInfo**
- MemoryWriteDelegate **writeDelegate**

8.49.1 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

8.50 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_)

8.50.1 Detailed Description

Used to set metadata for a table.

8.50.2 Member Function Documentation

8.50.2.1 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.

8.50.2.2 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.

8.50.2.3 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.

8.50.2.4 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.

8.50.2.5 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.

8.50.2.6 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.

8.50.2.7 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.

8.50.2.8 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.

8.50.2.9 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.

8.50.2.10 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.

8.50.2.11 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.

8.50.2.12 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.

8.50.2.13 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.

8.50.2.14 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.

8.50.2.15 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

8.51 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_)

8.51.1 Detailed Description

Used to set metadata for a table column.

8.51.2 Member Function Documentation

8.51.2.1 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.

8.51.2.2 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.

8.51.2.3 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.

8.51.2.4 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.

8.51.2.5 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.

8.51.2.6 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.

8.51.2.7 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.

8.51.2.8 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.

8.51.2.9 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.

8.51.2.10 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.

8.51.2.11 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

8.52 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.

8.52.1 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) 2017-2023 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) IRIS_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) IRIS_OVERRIDE
142     {
143         IrisC_CommunicationChannel channel;
144
145         channel.CommunicationChannel_version = 0;
146         channel.handleMessage_function       = &IrisCConnection::handleMessageToIrisInterface;
147         channel.handleMessage_context        = static_cast<void*>(iris_interface);
148
149         uint64_t channelId = IRIS_UINT64_MAX;
150
151         IrisErrorCode status = static_cast<IrisErrorCode>(IrisC_registerChannel(&channel, &channelId));
152
153         if (status != E_ok)
154         {
155             throw IrisErrorException(status);
156         }
157
158         return channelId;
159     }
160
165     virtual void unregisterIrisInterfaceChannel(uint64_t channelId) IRIS_OVERRIDE
166     {
167         IrisErrorCode status = static_cast<IrisErrorCode>(IrisC_unregisterChannel(channelId));
168
169         if (status != E_ok)
170         {
171             throw IrisErrorException(status);
172         }
173     }
174
179     virtual IrisErrorCode processAsyncMessages(bool waitForAMessage) IRIS_OVERRIDE
180     {
181         return static_cast<IrisErrorCode>(IrisC_processAsyncMessages(waitForAMessage));
182     }
183
188     virtual IrisInterface* getIrisInterface() IRIS_OVERRIDE
189     {
190         return &remote_interface;
191     }
192 };
193
194 NAMESPACE_IRIS_END
195
196 #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 <map>
#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 2015-2022 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/IrisInstance.h"
26
27 #include <map>
28 #include <memory>
29 #include <mutex>
30 #include <queue>
31 #include <thread>
32 #include <vector>
33 #if defined(__linux__) || defined(__APPLE__)
34 #include <csignal>
35 #include <sys/types.h>
36 #include <sys/wait.h>
37 #endif
38 #if defined(__linux__)
39 #include <sys/prctl.h>
40 #endif
41
42 NAMESPACE_IRIS_INTERNAL_START(service)
43 class IrisServiceTcpServer;
44 NAMESPACE_IRIS_INTERNAL_END
45
46 NAMESPACE_IRIS_START
47
48 class IrisClient
49     : public IrisInterface
50     , public impl::IrisProcessEventsInterface
51     , public IrisConnectionInterface
52 {
53 public:
55     IrisClient(const std::string& instName = std::string(), const std::string& connectionSpec =
      std::string())
56     {
57         init(IRIS_TCP_CLIENT, instName);
58         if (!connectionSpec.empty())
59         {
60             connect(connectionSpec);
61         }
62     }
63
65     IrisClient(const service::IrisServiceTcpServer*, const std::string& instName = std::string())
66     {
```

```
67          init(IRIS_SERVICE_SERVER, instName);
68      }
69
80      IrisClient(const std::string& hostname, uint16_t port, const std::string& instName = std::string())
81      {
82          init(IRIS_TCP_CLIENT, instName);
83          std::string ignored_error;
84          IrisErrorCode status = connect(hostname, port, port ? 1000 : 100, ignored_error);
85          if (status != E_ok)
86          {
87              throw IrisErrorExceptionString(status, "Failed to connect to Iris TCP server");
88          }
89      }
90
92      virtual ~IrisClient()
93      {
94          disconnect();
95
96          // Do not rely on destructor order. The socket_thread expects this
97          // object to be fully alive.
98          if (socket_thread)
99          {
100             socket_thread->terminate();
101         }
102
103         switch (mode)
104         {
105         case IRIS_TCP_CLIENT:
106             socketSet.removeSocket(&sock);
107             break;
108
109         case IRIS_SERVICE_SERVER:
110             socketSet.removeSocket(service_socket);
111             // remove service_socket TODO safer memory management
112             delete service_socket;
113             break;
114         }
115
116         iris::sleepMs(sleepOnDestructionMs);
117     }
118
123     void spawnAndConnect(const std::vector<std::string>& modelCommandLine, const std::string&
    additionalServerArgs = std::string(), const std::string& additionalClientArgs = std::string())
124     {
125 #ifdef _WIN32
126         (void)modelCommandLine;
127         (void)additionalServerArgs;
128         (void)additionalClientArgs;
129         if (modelCommandLine.size() < 1000000) // Hack: Disable spurious "unreachable code" warning in
    code calling spawnAndConnect() on Windows while we have not implemented this.
130         {
131             throw IrisErrorExceptionString(E_not_connected, "socketpair() connections not yet supported
    on Windows");
132         }
133 #else
134         // Increase verbose level? (connect() below does this, but is too late)
135         IrisCommaSeparatedParameters clientArgs(additionalClientArgs, "1");
136         setVerbose(unsigned(clientArgs.getUint("verbose", 0)), /*increaseOnly=*/true);
137
138         if (isConnected() || (childPid > 0))
139         {
140             disconnectAndWaitForChildToExit();
141         }
142
143         // Create socket pair.
144         int socketfd[2]; // We arbitrarily choose: 0=parent/client, 1=child/server
145         enum { CLIENT, SERVER };
146         if (socketpair(PF_LOCAL, SOCK_STREAM, 0, socketfd))
147         {
148             throw IrisErrorExceptionString(E_socket_error, "socketpair() failed");
149         }
150
151         lastExitStatus = -1;
152
153         // Fork.
154         childPid = fork();
155         if (childPid == 0)
156         {
157             // Child == server/model.
158             close(socketfd[CLIENT]);
159
160 #if defined(__linux__)
161             // Ask the kernel to kill us with SIGINT on parent thread termination.
162             // NOTE: Cleared on fork, but not on exec.
163             prctl(PR_SET_PDEATHSIG, SIGINT);
164 #endif
165
```

```
166             // Prepare args.
167             std::vector<std::string> args = modelCommandLine;
168             args.push_back("--iris-connect");
169             args.push_back("socketfd=" + std::to_string(socketfd[SERVER]) + "," + additionalServerArgs);
170             std::vector<const char *> cargs;
171             for (const std::string& s: args)
172             {
173                 cargs.push_back(s.c_str());
174             }
175             cargs.push_back(nullptr);
176 
177             // Start model. Replaces the currently running executable. Does not return on success.
178             execve(cargs[0], (char * const *)cargs.data(), environ);
179 
180             // execve() only returns on error.
181             close(socketfd[SERVER]);
182             throw IrisErrorExceptionString(E_not_connected, "execve() failed. Error launching model
    (command line: " + iris::joinString(args, " ") + ").");
183         }
184         else if (childPid < 0)
185         {
186             close(socketfd[CLIENT]);
187             close(socketfd[SERVER]);
188             childPid = 0;
189             throw IrisErrorExceptionString(E_not_connected, "fork() failed with errno=" +
    std::to_string(errno) + ".");
190         }
191         else
192         {
193             if (verbose)
194             {
195                 log.info("IrisClient::spawnAndConnect(): Spawned child process %d.\n", int(childPid));
196             }
197 
198             // Parent == client/debugger.
199             close(socketfd[SERVER]);
200 
201             try
202             {
203                 // Connect to model.
204                 connect("socketfd=" + std::to_string(socketfd[CLIENT]) + "," + additionalClientArgs);
205             }
206             catch (...)
207             {
208                 // connect() already closed the socket on error.
209 
210                 // Issue SIGINT and then SIGKILL to terminate child.
211                 disconnectAndWaitForChildToExit(0);
212                 throw;
213             }
214         }
215 #endif
216     }
217 
231     bool disconnectAndWaitForChildToExit(double timeoutInMs = 5000, double timeoutInMsAfterSigInt =
    5000, double timeoutInMsAfterSigKill = 5000)
232     {
233         if (verbose)
234         {
235             log.info("IrisClient::disconnectAndWaitForChildToExit(timeoutInMs=%.0f,
    timeoutInMsAfterSigInt=%.0f, timeoutInMsAfterSigKill=%.0f)\n", timeoutInMs, timeoutInMsAfterSigInt,
    timeoutInMsAfterSigKill);
236         }
237 
238         // Disconnect.
239         IrisErrorCode error = disconnect();
240         if (error)
241         {
242             throw IrisErrorExceptionString(E_not_connected, "disconnect() failed.");
243         }
244 
245 #ifdef _WIN32
246         (void)timeoutInMs;
247         (void)timeoutInMsAfterSigInt;
248         (void)timeoutInMsAfterSigKill;
249         throw IrisErrorExceptionString(E_not_implemented, "socketpair() connections not yet supported on
    Windows.");
250 #else
251         if (childPid == 0)
252         {
253             return true;
254         }
255 
256         if (!floatEqual(timeoutInMs, 0.0))
257         {
258             // Wait for child process to exit for timeoutInMs.
259             if (waitpidWithTimeout(childPid, &lastExitStatus, 0, timeoutInMs))
```

```
260             {
261                 childPid = 0;
262                 return true;
263             }
264         }
265 
266         if (!floatEqual(timeoutInMsAfterSigInt, 0.0))
267         {
268             // Send SIGINT and wait for timeoutInMsAfterSigInt.
269             if (verbose)
270             {
271                 log.info("IrisClient::disconnectAndWaitForChildToExit(): Sending SIGINT to child %d.\n",
    int(childPid));
272             }
273             if (kill(childPid, SIGINT) < 0)
274             {
275                 throw IrisErrorExceptionString(E_not_connected, "kill(SIGINT) failed with errno=" +
    std::to_string(errno) + ".");
276             }
277             if (waitpidWithTimeout(childPid, &lastExitStatus, 0, timeoutInMsAfterSigInt))
278             {
279                 childPid = 0;
280                 return true;
281             }
282         }
283 
284         if (!floatEqual(timeoutInMsAfterSigKill, 0.0))
285         {
286             // Send SIGKILL and wait for timeoutInMsAfterSigKill.
287             if (verbose)
288             {
289                 log.info("IrisClient::disconnectAndWaitForChildToExit(): Sending SIGKILL to child
    %d.\n", int(childPid));
290             }
291             if (kill(childPid, SIGKILL) < 0)
292             {
293                 throw IrisErrorExceptionString(E_not_connected, "kill(SIGKILL) failed with errno=" +
    std::to_string(errno) + ".");
294             }
295             if (waitpidWithTimeout(childPid, &lastExitStatus, 0, timeoutInMsAfterSigKill))
296             {
297                 childPid = 0;
298                 return true;
299             }
300         }
301 
302         // Child did not exit so far.
303         if (verbose)
304         {
305             log.info("IrisClient::disconnectAndWaitForChildToExit(): Child %d did not exit.\n",
    int(childPid));
306         }
307         return false;
308 #endif
309     }
310 
311 #ifndef _WIN32
315     bool waitpidWithTimeout(pid_t pid, int* status, int options, double timeoutInMs)
316     {
317         if (verbose)
318         {
319             log.info("IrisClient::waitpidWithTimeout(): Waiting %.1f ms for child %d to exit ...\n",
    timeoutInMs, int(pid));
320         }
321 
322         double endTime = getTimeInSec() + timeoutInMs / 1000.0;
323         if (timeoutInMs < 0)
324         {
325             endTime += 1e100;
326         }
327 
328         // Wait for child to exit.
329         while (getTimeInSec() < endTime)
330         {
331             pid_t ret = waitpid(pid, status, options | WNOHANG);
332             if (ret == pid)
333             {
334                 if (verbose)
335                 {
336                     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));
337                 }
338                 return true; // Child exited.
339             }
340             if (ret < 0)
341             {
```

```
342                 throw IrisErrorExceptionString(E_not_connected, "waitpid() failed with errno=" +
    std::to_string(errno) + ".");
343             }
344             if (ret > 0)
345             {
346                 throw IrisErrorExceptionString(E_not_connected, "waitpid() returned unexpected pid=" +
    std::to_string(pid) + ".");
347             }
348             assert(ret == 0);
349 
350             sleepMs(20);
351         }
352 
353         return false; // Timeout.
354     }
355 #endif
356 
357 #ifndef _WIN32
359     pid_t getChildPid() const
360     {
361         return childPid;
362     }
363 #endif
364 
366     int getLastExitStatus() const { return lastExitStatus; }
367 
369     const std::string connectionHelpStr =
370         "Supported connection types:\n"
371         "tcp[=HOST][,port=PORT][,timeout=T]\n"
372         "  Connect to an Iris TCP server on HOST:PORT.\n"
373         "  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"
374         "  T is the connection timeout in ms (defaults to 100 if PORT==0, else 1000).\n"
375         "\n"
376         "socketfd=FD[,timeout=T]\n"
377         "  Use socket file descriptor FD as an established UNIX domain socket connection.\n"
378         "  T is the timeout for the Iris handshake in ms.\n"
379         "\n"
380         "General parameters:\n"
381         "  verbose=N: Increase verbose level of IrisClient to level N (0..3).\n";
382 
391     void connect(const std::string& connectionSpec)
392     {
393         if (verbose)
394         {
395             log.info("IrisClient::connect(%s)\n", connectionSpec.c_str());
396         }
397 
398         IrisCommaSeparatedParameters params(connectionSpec, "1");
399 
400         // Emit help message?
401         if (params.have("help"))
402         {
403             throw IrisErrorExceptionString(E_help_message, connectionHelpStr);
404         }
405 
406         // Increase verbose level?
407         setVerbose(unsigned(params.getUint("verbose", 0)), /*increaseOnly=*/true);
408 
409         // Validate connection type.
410         if (unsigned(params.have("tcp")) + unsigned(params.have("socketfd")) != 1)
411         {
412             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.");
413         }
414 
415         if (params.have("tcp"))
416         {
417             std::string hostname = params.getStr("tcp");
418             if (hostname == "1")
419             {
420                 hostname = "localhost";
421             }
422             uint16_t port = uint16_t(params.getUint("port", hostname == "localhost" ? 0 : 7100));
423             unsigned timeoutInMs = unsigned(params.getUint("timeout", port == 0 ? 100 : 1000));
424             if (params.haveUnusedParameters())
425             {
426                 throw IrisErrorExceptionString(E_not_connected, params.getUnusedParametersMessage("Error
    in 'tcp' connection parameters: "));
427             }
428             std::string errorResponse;
429             IrisErrorCode status = connect(hostname, port, timeoutInMs, errorResponse);
430             if (status != E_ok)
431             {
432                 throw IrisErrorExceptionString(status, errorResponse);
433             }
```

```
434         }
435
436         if (params.have("socketfd"))
437         {
438             SocketFd socketfd = SocketFd(params.getUint("socketfd"));
439             unsigned timeoutInMs = unsigned(params.getUint("timeout", 1000));
440             if (params.haveUnusedParameters())
441             {
442                 throw IrisErrorExceptionString(E_not_connected, params.getUnusedParametersMessage("Error
    in 'socketfd' connection parameters: "));
443             }
444             connectSocketFd(socketfd, timeoutInMs);
445         }
446     }
447
452     IrisErrorCode connect(const std::string& hostname, uint16_t port, unsigned timeoutInMs, std::string&
    errorResponseOut)
453     {
454         assert(mode == IRIS_TCP_CLIENT);
455
456         if (verbose)
457             log.info("IrisClient::connect(hostname=%s, port=%u, timeout=%u) enter\n", hostname.c_str(),
    port, timeoutInMs);
458
459         // Already connected?
460         IrisErrorCode error = E_ok;
461         if (adapter.isConnected() || sock.isConnected())
462         {
463             error = E_already_connected;
464             goto done;
465         }
466
467         // hostname==localhost and port==0 means port scan.
468         if ((hostname == "localhost") && (port == 0))
469         {
470             const uint16_t startport = 7100;
471             const uint16_t endport = 7109;
472             for (port = startport; port <= endport; port++)
473             {
474                 std::string errorMessage;
475                 if (connect(hostname, port, timeoutInMs, errorResponseOut) == iris::E_ok)
476                     return E_ok;
477             }
478             errorResponseOut = "No Iris TCP server found on ports " + std::to_string(startport) + ".." +
    std::to_string(endport) + "\n";
479             error = E_not_connected;
480             goto done;
481         }
482
483         if (!sock.isCreated())
484         {
485             sock.create();
486             sock.setNonBlocking();
487
488             // Unblock a potentially blocked worker thread which so far is waiting indefinitely
489             // on 'no socket'. This thread will block again on the socket we just created.
490             socketSet.stopWaitForEvent();
491         }
492
493         // Connect to server.
494         error = sock.connect(hostname, port, timeoutInMs);
495         if (error != E_ok)
496         {
497             errorResponseOut = "Error connecting to " + hostname + ":" + std::to_string(port);
498             sock.close();
499             goto done;
500         }
501
502         // Initialize client.
503         error = initClient(timeoutInMs, errorResponseOut);
504         if (error == E_ok)
505         {
506             connectionStr = hostname + ":" + std::to_string(port);
507         }
508         else
509         {
510             disconnect();
511         }
512
513         // Return error code (if any).
514     done:
515         if (verbose)
516             log.info("IrisClient::connect() leave (%s)\n", irisErrorCodeCStr(error));
517         return error;
518     }
519
523     void connectSocketFd(SocketFd socketfd, unsigned timeoutInMs = 1000)
```

```
524     {
525         assert(mode == IRIS_TCP_CLIENT);
526 
527         if (verbose)
528             log.info("IrisClient::connectSocketFd(socketfd=%llu, timeout=%u)\n", (long long)socketfd,
    timeoutInMs);
529 
530         // Already connected?
531         std::string errorResponse;
532         IrisErrorCode error = E_ok;
533         if (adapter.isConnected() || sock.isConnected())
534         {
535             throw IrisErrorExceptionString(E_already_connected, "Already connected.");
536         }
537 
538         sock.setSocketFd(socketfd);
539         sock.setNonBlocking();
540 
541         // Unblock a potentially blocked worker thread which so far is waiting indefinitely
542         // on 'no socket'. This thread will block again on the socket we just created.
543         socketSet.stopWaitForEvent();
544 
545         // Initialize client.
546         error = initClient(timeoutInMs, errorResponse);
547         if (error != E_ok)
548         {
549             disconnect();
550             throw IrisErrorExceptionString(error, errorResponse);
551         }
552 
553         connectionStr = "(connected via socketfd)";
554     }
555 
559     IrisErrorCode disconnect()
560     {
561         if (verbose)
562         {
563             log.info("IrisClient::disconnect()\n");
564         }
565 
566         // Tell IrisInstance to stop sending requests to us.
567         // All Iris calls (including the inevitable final
568         // instanceRegistry_unregisterInstance()) will return
569         // E_not_connected from now on.
570         irisInstance.setConnectionInterface(nullptr);
571 
572         connectionStr = "(not connected)";
573 
574         if (mode != IRIS_TCP_CLIENT)
575         {
576             return E_ok;
577         }
578 
579         // We just close the TCP connection. This is a first-class operation which always must be
    handled gracefully by the server.
580         // The server needs to do all cleanup automatically.
581         IrisErrorCode errorCode = E_ok;
582         if (adapter.isConnected())
583             errorCode = adapter.closeConnection();
584         if (sock.isConnected())
585         {
586             if (errorCode != E_ok)
587                 sock.close();
588             else
589                 errorCode = sock.close();
590         }
591 
592         // Wake up processing thread since there is no point to wait on a closed socket.
593         socketSet.stopWaitForEvent();
594 
595         return errorCode;
596     }
597 
599     bool isConnected() const
600     {
601         return adapter.isConnected();
602     }
603 
605     IrisInterface* getSendingInterface()
606     {
607         return this;
608     }
609 
612     void setInstanceName(const std::string& instName)
613     {
614         if (irisInstance.isRegistered())
615         {
```

```
616             throw IrisErrorExceptionString(E_instance_already_registered, "IrisClient::setInstanceName()
    must be called before connect().");
617         }
618         irisInstanceInstName = instName;
619     }
620
623     IrisInstance& getIrisInstance() { return irisInstance; }
624
627     void setSleepOnDestructionMs(uint64_t sleepOnDestructionMs_)
628     {
629         sleepOnDestructionMs = sleepOnDestructionMs_;
630     }
631
632
633     // --- IrisProcessEventsInterface implementation ---
634
651     virtual void processEvents() override
652     {
653         if (verbose >= 2)
654             log.info("IrisClient::processEvents() enter\n");
655
656         // in IRIS_SERVICE_SERVER mode, the adapter should work as server and hence call
657         // function processEventsServer()
658         switch (mode)
659         {
660         case IRIS_TCP_CLIENT:
661             adapter.processEventsClient();
662             break;
663         case IRIS_SERVICE_SERVER:
664             adapter.processEventsServer();
665             break;
666         }
667
668         if (verbose >= 2)
669             log.info("IrisClient::processEvents() leave\n");
670     }
671
675     virtual void waitForEvent() override
676     {
677         if (verbose >= 2)
678             log.info("IrisClient::waitForEvent() enter\n");
679         socketSet.waitForEvent(1000);
680         if (verbose >= 2)
681             log.info("IrisClient::waitForEvent() leave\n");
682     }
683
686     virtual void stopWaitForEvent() override
687     {
688         if (verbose)
689             log.info("IrisClient::stopWaitForEvent()\n");
690         socketSet.stopWaitForEvent();
691     }
692
694     void setPreferredSendingFormat(impl::IrisRpcAdapterTcp::Format p)
695     {
696         adapter.setPreferredSendingFormat(p);
697     }
698
700     impl::IrisRpcAdapterTcp::Format getEffectiveSendingFormat() const
701     {
702         return adapter.getEffectiveSendingFormat();
703     }
704
706     void setVerbose(unsigned level, bool increaseOnly = false)
707     {
708         if (increaseOnly && (level < verbose))
709         {
710             return;
711         }
712
713         verbose = level;
714         if (verbose)
715             log.info("IrisClient: verbose logging enabled (level %d)\n", verbose);
716         if (mode == IRIS_TCP_CLIENT)
717         {
718             sock.setVerbose(verbose);
719         }
720         socketSet.setVerbose(verbose);
721         if (verbose)
722         {
723             log.setIrisMessageLogLevelFlags(IrisLogger::TIMESTAMP);
724         }
725     }
726
728     void setIrisMessageLogLevel(unsigned level) { irisMessageLogLevel = level;
    log.setIrisMessageLogLevel(irisMessageLogLevel); }
729
```

```
731     std::string getConnectionStr() const { return connectionStr; }
732
733 private:
735     enum Mode
736     {
737         IRIS_TCP_CLIENT,
738         IRIS_SERVICE_SERVER
739     };
740
741     // Shared code for constructors in client mode.
742     void init(Mode mode_, const std::string& instName)
743     {
744         log.setLogContext("IrisTC");
745         mode = mode_;
746
747         // Set instance name of contained IrisInstance.
748         if (instName.empty())
749         {
750             setInstanceName("client.IrisClient");
751         }
752         else
753         {
754             setInstanceName(instName);
755         }
756
757         // Enable verbose logging?
758         setVerbose(static_cast<unsigned>(getEnvU64("IRIS_TCP_CLIENT_VERBOSE")), true);
759         irisMessageLogLevel = unsigned(getEnvU64("IRIS_TCP_CLIENT_LOG_MESSAGES"));
760         log.setIrisMessageLogLevel(irisMessageLogLevel);
761         log.setIrisMessageGetInstNameFunc([&](InstanceId instId){ return getInstName(instId); });
762
763         if (mode == IRIS_TCP_CLIENT)
764         {
765             socketSet.addSocket(&sock);
766         }
767         sendingInterface = adapter.getSendingInterface();
768
769         // Intercept all calls to the global instance since we must modify
    instanceRegistry_registerInstance() and
770         // instanceRegistry_unregisterInstance() and their responses.
771         instIdToInterface.push_back(&globalInstanceSendingInterface); // This must be index 0 in the
    vector (instId 0 == global instance).
772
773         if (mode == IRIS_SERVICE_SERVER)
774         {
775             socket_thread = std::unique_ptr<impl::IrisProcessEventsThread>(new
    impl::IrisProcessEventsThread(this, "TcpSocket"));
776         }
777     }
778
780     IrisErrorCode initClient(unsigned timeoutInMs, std::string& errorResponseOut)
781     {
782         assert(mode == IRIS_TCP_CLIENT);
783
784         // Initialize IrisRpcAdapterTcp.
785         try
786         {
787             adapter.initClient(&sock, &socketSet, &receivingInterface, verbose);
788         }
789         catch (const IrisErrorException& e)
790         {
791             if (e.getMessage().empty())
792             {
793                 throw IrisErrorExceptionString(e.getErrorCode(), "Client: Error connecting to server
    socket.");
794             }
795             else
796             {
797                 throw;
798             }
799         }
800
801         // Handshake.
802         IrisErrorCode error = adapter.handshakeClient(errorResponseOut, timeoutInMs);
803
804         // Start a thread to process incoming data in the background.
805         socket_thread = std::unique_ptr<impl::IrisProcessEventsThread>(new
    impl::IrisProcessEventsThread(this, "TcpSocket"));
806
807         // Initialize IrisInstance.
808         irisInstance.setConnectionInterface(this);
809         irisInstance.registerInstance(irisInstanceInstName, iris::IrisInstance::UNIQUIFY |
    iris::IrisInstance::THROW_ON_ERROR);
810
811         return error;
812     }
813
```

```
817     virtual void irisHandleMessage(const uint64_t* message) override
818     {
819         // Log message?
820         if (irisMessageLogLevel)
821         {
822             log.irisMessage(message);
823         }
824 
825         // This calls one of these:
826         // - this->globalInstanceSendingInterface_irisHandleMessage(); (for requests, instId == 0)
827         // - Iris interface of a local instance (if a local instance talks to a local instance)
828         // - sendingInterface (to send message to server using TCP)
829         findInterface(IrisU64JsonReader::getInstId(message))->irisHandleMessage(message);
830     }
831 
834     void globalInstanceSendingInterface_irisHandleMessage(const uint64_t* message)
835     {
836         // This is only ever called for instId == 0.
837         assert(IrisU64JsonReader::getInstId(message) == 0);
838         assert(IrisU64JsonReader::isRequestOrNotification(message));
839 
840         // Decode request.
841         IrisU64JsonReader          r(message);
842         IrisU64JsonReader::Request req    = r.openRequest();
843         std::string                method = req.getMethod();
844 
845         if (method == "instanceRegistry_registerInstance")
846         {
847             RequestId requestId = req.getRequestId();
848 
849             // We received an instanceRegistry_registerInstance() request from a local instance:
850             // - Create a new request id which is unique to this request for this TCP channel. (This is
    not required to be globally unique.)
851             // - Allocate an ongoingInstanceRegistryCalls slot for this new request id and remember the
    original request id and params.channelId in it.
852             // - Modify request id of request to the new request id so we can recognize the response
    later.
853             // - Send modified request.
854 
855             // Create a new request id which is unique to this request for this TCP channel. (This is
    not required to be globally unique.)
856             RequestId newRequestId = generateNewRequestIdForRegisterInstanceCall();
857 
858             // Get channelId.
859             uint64_t channelId = IRIS_UINT64_MAX;
860             if (!req.paramOptional(ISTR("channelId"), channelId))
861             {
862                 // Strange. 'params.channelId' is missing. This should never happen.
863                 log.error(
864                     "IrisClient::receivingInterface_irisHandleMessage():"
865                     " Received instanceRegistry_registerInstance() request without channelId
    parameter:\n%s\n",
866                     messageToString(message).c_str());
867                 goto send;
868             }
869 
870             {
871                 std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
872                 // Allocate an ongoingInstanceRegistryCalls slot for this new request id and remember
    the
873                 // original request id and params.channelId in it.
874                 ongoingInstanceRegistryCalls[newRequestId] = OngoingInstanceRegistryCallEntry(method,
875                                                                                               requestId,
876 
    channelId);
877             }
878 
879             // Create a modified reaquest that:
880             //  - sets the new request id so we can recognize the response later.
881             //  - removes the channelId parameter (it only has meaning in-process)
882             IrisU64JsonReader original_message(message);
883             IrisU64JsonWriter modified_message;
884 
885             {
886                 IrisU64JsonReader::Request original_req = original_message.openRequest();
887 
888                 IrisU64JsonWriter::Request new_req =
    modified_message.openRequest(original_req.getMethod(),
889 
    original_req.getInstId());
890                 new_req.setRequestId(newRequestId);
891 
892                 std::string param;
893                 while (original_req.readNextParam(param))
894                 {
895                     if ((param == "channelId") || (param == "instId"))
896                     {
```

```
897                         // Skip the params we want to remove (channelId)
898                         // and skip instId too because that will have already been filled in.
899                         // skip over the value to the next parameter
900                         original_message.skip();
901                     }
902                     else
903                     {
904                         new_req.paramSlow(param);
905 
906                         // Pass through the original value
907                         IrisValue value;
908                         persist(original_message, value);
909                         persist(modified_message, value);
910                     }
911                 }
912             }
913 
914             // Send modified request.
915             sendingInterface->irisHandleMessage(modified_message.getMessage());
916             return;
917         }
918         else if (method == "instanceRegistry_unregisterInstance")
919         {
920             // We received an instanceRegistry_unregisterInstance() request from a local instance:
921             // - Allocate an ongoingInstanceRegistryCalls slot for the request id and remember the instId of the unregistered instance in it.
922             // - Send request unmodified.
923 
924             // Get params.aInstId.
925             InstanceId aInstId = IRIS_UINT64_MAX;
926             if (!req.paramOptional(ISTR("aInstId"), aInstId))
927             {
928                 // Strange. 'params.aInstId' is missing. This should never happen.
929                 log.error(
930                     "IrisClient::receivingInterface_irisHandleMessage():"
931                     " Received instanceRegistry_unregisterInstance() request without aInstId parameter:\n%s\n",
932                     messageToString(message).c_str());
933                 goto send;
934             }
935 
936             if (!req.isNotification())
937             {
938                 RequestId requestId = req.getRequestId();
939 
940                 if (aInstId == getCallerInstId(requestId))
941                 {
942                     std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
943                     // There will be a response to this request so we need to remember the interface to send it to.
944                     // Allocate an ongoingInstanceRegistryCalls slot for the request id and remember the instId of the unregistered instance in it.
945                     ongoingInstanceRegistryCalls[requestId] = OngoingInstanceRegistryCallEntry(method, aInstId);
946                     goto send;
947                 }
948             }
949 
950             // There will be no more communication to the instance being unregistered.
951             // Remove instance from instIdToInterface.
952             assert(aInstId < InstanceId(instIdToInterface.size()));
953             // 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.
954             instIdToInterface[aInstId] = sendingInterface;
955 
956             // Intended fallthrough to send original request.
957         }
958         else if (method == "instanceRegistry_getList")
959         {
960             // We received an instanceRegistry_getList() request from a local instance:
961             // - We want to remember/snoop all returned instance names we get in the response (for logging).
962             // - Allocate an ongoingInstanceRegistryCalls slot for the request id in order to recognize the response.
963             // - Send request unmodified.
964 
965             if (!req.isNotification())
966             {
967                 RequestId requestId = req.getRequestId();
968                 std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
969                 ongoingInstanceRegistryCalls[requestId] = OngoingInstanceRegistryCallEntry(method);
970             }
971 
972             // Intended fallthrough to send original request.
973         }
974 
975      send:
```

```
976         // Send original message.
977         sendingInterface->irisHandleMessage(message);
978     }
979
983     void receivingInterface_irisHandleResponse(const uint64_t* message)
984     {
985         {
986             std::lock_guard<std::mutex> lock(ongoingInstanceRegistryCallsMutex);
987
988             if (!ongoingInstanceRegistryCalls.empty())
989             {
990                 // Slow path is only used while a instanceRegistry_registerInstance() or
    instanceRegistry_unregisterInstance()
991                 // call is ongoing. This is usually only the case at startup and shutdown.
992
993                 // We need to check whether this is the response to either
994                 // instanceRegistry_registerInstance() or
995                 // instanceRegistry_unregisterInstance() or
996                 // any other response.
997
998                 // Decode response.
999                 IrisU64JsonReader           r(message);
1000                 IrisU64JsonReader::Response resp = r.openResponse();
1001                 RequestId requestId = resp.getRequestId();
1002
1003                 // Check whether this is a response to one of our pending requests.
1004                 OngoingInstanceRegistryCallMap::iterator i =
    ongoingInstanceRegistryCalls.find(requestId);
1005                 if (i == ongoingInstanceRegistryCalls.end())
1006                 {
1007                     goto send; // None of the pending responses. Handle in the normal way.
1008                 }
1009
1010                 if (i->second.method == "instanceRegistry_registerInstance")
1011                 {
1012                     // This is a response to a previous instanceRegistry_registerInstance() call:
1013
1014                     IrisInterface* responseIfPtr = channel_registry.getChannel(i->second.channelId);
1015
1016                     if (resp.isError())
1017                     {
1018                         // The call failed, pass on the message.
1019                         responseIfPtr->irisHandleMessage(message);
1020                     }
1021                     else
1022                     {
1023                         // The call succeeded:
1024                         // - add new instId to our local instance registry
1025                         // - translate request id back to the original request id
1026                         // - send this modified response to the caller
1027                         // - erase this entry in ongoingInstanceRegistryCalls
1028
1029                         // Add instance to instIdToInterface.
1030                         InstanceId newInstId;
1031                         if (!resp.getResultReader().openObject().memberOptional(ISTR("instId"),
    newInstId))
1032                         {
1033                             // Strange. 'result.instId' is missing. This should never happen.
1034                             log.error(
1035                                 "IrisClient::receivingInterface_irisHandleResponse():"
1036                                 " Received instanceRegistry_registerInstance() response without
    result.instId:\n%s\n",
1037                                 messageToString(message).c_str());
1038                         }
1039                         else
1040                         {
1041                             // This is a valid response for instanceRegistry_registerInstance(): Enter
    newInstId into instIdToInterface.
1042                             findInterface(newInstId);
1043                             instIdToInterface[newInstId] = responseIfPtr;
1044                         }
1045
1046                         // Remember instance name.
1047                         std::string newInstName;
1048                         if (resp.getResultReader().openObject().memberOptional(ISTR("instName"),
    newInstName))
1049                         {
1050                             setInstName(newInstId, newInstName);
1051                         }
1052
1053                         // Translate the id back to the id of the original request and use the
    responseIfPtr to send the response.
1054                         IrisU64JsonWriter modifiedMessageWriter;
1055                         modifiedMessageWriter.copyMessageAndModifyId(message, i->second.id);
1056
1057                         // Log message?
1058                         if (irisMessageLogLevel)
```

```
1059                        {
1060                            log.irisMessage(modifiedMessageWriter.getMessage());
1061                        }
1062
1063                        responseIfPtr->irisHandleMessage(modifiedMessageWriter.getMessage());
1064                    }
1065
1066                    // Remove ongoingInstanceRegistryCalls entry now that we have seen the response.
1067                    ongoingInstanceRegistryCalls.erase(i);
1068                    return;
1069                }
1070                else if (i->second.method == "instanceRegistry_unregisterInstance")
1071                {
1072                    // This is a response to a previous instanceRegistry_unregisterInstance() call:
1073                    // - remove this instId from our local instance registry
1074                    // - remove this entry from ongoingInstanceRegistryCalls
1075                    // - send response to caller
1076
1077                    InstanceId aInstId = i->second.id;
1078
1079                    // Remeber the old response interface in case we need it after we override it
1080                    IrisInterface* aInst_responseIf = instIdToInterface[aInstId];
1081
1082                    // Remove instance from instIdToInterface.
1083                    assert(aInstId < InstanceId(instIdToInterface.size()));
1084                    // 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.
1085                    instIdToInterface[aInstId] = sendingInterface;
1086                    setInstName(aInstId, ""); // IrisLogger will generate a default name for unknown
    instance ids.
1087                    // Remove ongoingInstanceRegistryCalls entry.
1088                    ongoingInstanceRegistryCalls.erase(i);
1089
1090                    if (aInstId == resp.getInstId())
1091                    {
1092                        // An instance unregistered itself so we need to call it directly rather than
1093                        // go through the normal message handler because we just set that to forward
1094                        // messages to this instId to the server.
1095                        aInst_responseIf->irisHandleMessage(message);
1096                        return;
1097                    }
1098
1099                    // Intended fallthrough to irisHandleMessage(message).
1100                }
1101                else if (i->second.method == "instanceRegistry_getList")
1102                {
1103                    // This is a response to a previous instanceRegistry_getList() call:
1104                    // - remember all instance names (for logging)
1105                    // - send response to caller
1106
1107                    // Remove ongoingInstanceRegistryCalls entry.
1108                    ongoingInstanceRegistryCalls.erase(i);
1109                    try
1110                    {
1111                        // Peek into instance list. We do not care whether this is just
1112                        // a subset of all instances or not. We take what we can get.
1113                        std::vector<InstanceInfo> instanceInfoList;
1114                        resp.getResult(instanceInfoList);
1115                        for (const auto& instanceInfo: instanceInfoList)
1116                        {
1117                            setInstName(instanceInfo.instId, instanceInfo.instName);
1118                        }
1119                    }
1120                    catch(const IrisErrorException&)
1121                    {
1122                        // Silently ignore bogus responses. The caller will handle the error.
1123                    }
1124                    // Intended fallthrough to irisHandleMessage(message).
1125                }
1126            }
1127        }
1128
1129    send:
1130        // Handle response in the normal way.
1131        irisHandleMessage(message);
1132    }
1133
1138    RequestId generateNewRequestIdForRegisterInstanceCall()
1139    {
1140        return nextInstIdForRegisterInstanceCall++;
1141    }
1142
1148    IrisInterface* findInterface(InstanceId instId)
1149    {
1150        if (instId >= IrisMaxTotalInstances)
1151        {
1152            log.error("IrisClient::findInterface(instId=0x%08x): got ridiculously high instId",
```

```
    int(instId));
1153            return sendingInterface;
1154        }
1155        if (instId >= InstanceId(instIdToInterface.size()))
1156        {
1157            instIdToInterface.resize(instId + 100, sendingInterface);
1158        }
1159        return instIdToInterface[instId];
1160    }
1161
1163    class GlobalInstanceSendingInterface : public IrisInterface
1164    {
1165    public:
1166        GlobalInstanceSendingInterface(IrisClient* parent_)
1167            : parent(parent_)
1168        {
1169        }
1170
1174        virtual void irisHandleMessage(const uint64_t* message) override
1175        {
1176            if (IrisU64JsonReader::isRequestOrNotification(message))
1177            {
1178                // Intercept requests to the global instance so we can snoop on
1179                // calls to instanceRegistry_registerInstance()
1180                parent->globalInstanceSendingInterface_irisHandleMessage(message);
1181            }
1182            else
1183            {
1184                // This is called for responses sent from clients to the global instance.
1185                // Simply forward them as usual. Nothing to intercept.
1186                parent->sendingInterface->irisHandleMessage(message);
1187            }
1188        }
1189
1190    private:
1191        IrisClient* const parent;
1192    };
1193
1195    class ReceivingInterface : public IrisInterface
1196    {
1197    public:
1198        ReceivingInterface(IrisLogger& log_, IrisClient* parent_)
1199            : parent(parent_)
1200            , log(log_)
1201        {
1202        }
1203
1205        virtual void irisHandleMessage(const uint64_t* message) override
1206        {
1207            InstanceId instId = IrisU64JsonReader::getInstId(message);
1208
1209            if (instId >= InstanceId(instId_to_thread_id.size()))
1210            {
1211                // We do not have an entry for this instance therefore
1212                // we have not been asked to marshal requests to a specific
1213                // thread and should use the default.
1214                // Todo: Remove once IrisMessageQueue and IrisProcessEventsThread are gone
1215                setHandlerThread(instId, getDefaultThreadId());
1216            }
1217
1218            // Todo: Refactor once IrisMessageQueue and IrisProcessEventsThread are gone
1219            std::thread::id thread_id = instId_to_thread_id[instId];
1220            if (thread_id == std::this_thread::get_id())
1221            {
1222                // Message has already been marshalled, forward on
1223                if (IrisU64JsonReader::isRequestOrNotification(message))
1224                {
1225                    parent->irisHandleMessage(message);
1226                }
1227                else
1228                {
1229                    parent->receivingInterface_irisHandleResponse(message);
1230                }
1231            }
1232            else
1233            {
1234                message_queue.push(message, thread_id);
1235            }
1236        }
1237
1238        void setHandlerThread(InstanceId instId, std::thread::id thread_id)
1239        {
1240            if (instId >= IrisMaxTotalInstances)
1241            {
1242                log.error(
1243                    "IrisClient::ReceivingInterface::setHandlerThread(instId=0x%08x):"
1244                    " got ridiculously high instId",
```

```
1245                     int(instId));
1246             }
1247             else if (instId >= InstanceId(instId_to_thread_id.size()))
1248             {
1249                 instId_to_thread_id.resize(instId + 100, getDefaultThreadId());
1250             }
1251
1252             instId_to_thread_id[instId] = thread_id;
1253         }
1254
1255         IrisErrorCode processMessagesForCurrentThread(bool waitForAMessage)
1256         {
1257             if (waitForAMessage)
1258             {
1259                 IrisErrorCode code = message_queue.waitForMessageForCurrentThread();
1260                 if (code != E_ok)
1261                 {
1262                     return code;
1263                 }
1264             }
1265             message_queue.processRequestsForCurrentThread();
1266
1267             return E_ok;
1268         }
1269
1270     private:
1272         std::thread::id getDefaultThreadId()
1273         {
1274             return process_events_thread.getThreadId();
1275         }
1276
1277         IrisClient* const parent;
1278
1280         impl::IrisMessageQueue message_queue{this};
1281
1283         std::vector<std::thread::id> instId_to_thread_id;
1284
1285         IrisLogger& log;
1286
1289         impl::IrisProcessEventsThread process_events_thread{&message_queue, "ClientMsgHandlr"};
1290     };
1291
1292 public: // IrisConnectionInterface
1293     virtual uint64_t registerIrisInterfaceChannel(IrisInterface* iris_interface) override
1294     {
1295         return channel_registry.registerChannel(iris_interface);
1296     }
1297
1298     virtual void unregisterIrisInterfaceChannel(uint64_t channelId) override
1299     {
1300         IrisInterface* if_to_remove = channel_registry.getChannel(channelId);
1301
1302         std::vector<InstanceId> instIds_for_channel;
1303
1304         for (size_t i = 0; i < instIdToInterface.size(); i++)
1305         {
1306             if (instIdToInterface[i] == if_to_remove)
1307             {
1308                 InstanceId instId = InstanceId(i);
1309                 instIds_for_channel.push_back(instId);
1310             }
1311         }
1312         if (instIds_for_channel.size() > 0)
1313         {
1314             // Create an instance to call instanceRegistry_unregisterInstance() with.
1315             IrisInstance instance_killer(this, "framework.IrisClient.instance_killer",
1316                                          IrisInstance::UNIQUIFY);
1317             for (InstanceId instId : instIds_for_channel)
1318             {
1319                 instance_killer.irisCall().instanceRegistry_unregisterInstance(instId);
1320             }
1321         }
1322
1323         channel_registry.unregisterChannel(channelId);
1324     }
1325
1326     virtual IrisErrorCode processAsyncMessages(bool waitForAMessage) override
1327     {
1328         return receivingInterface.processMessagesForCurrentThread(waitForAMessage);
1329     }
1330
1331     virtual IrisInterface* getIrisInterface() override
1332     {
1333         return this;
1334     }
1335
1336     uint64_t registerChannel(IrisC_CommunicationChannel* channel)
```

```
1337     {
1338         return channel_registry.registerChannel(channel);
1339     }
1340
1341     void unregisterChannel(uint64_t channelId)
1342     {
1343         channel_registry.unregisterChannel(channelId);
1344     }
1345
1346     // function called by class IrisPlugin
1347     uint64_t registerChannel(IrisC_CommunicationChannel* channel, const ::std::string& path)
1348     {
1349         (void) path;
1350         return channel_registry.registerChannel(channel);
1351     }
1352
1353 public:
1356     void loadPlugin(const std::string& plugin_name)
1357     {
1358         assert(mode == IRIS_SERVICE_SERVER);
1359         assert(plugin == nullptr);
1360         plugin = std::unique_ptr<impl::IrisPlugin<IrisClient>>(new impl::IrisPlugin<IrisClient>(this,
      plugin_name));
1361     }
1362
1363     void unloadPlugin()
1364     {
1365         assert(mode == IRIS_SERVICE_SERVER);
1366         plugin = nullptr;
1367     }
1368
1375     void initServiceServer(impl::IrisTcpSocket* socket_)
1376     {
1377         assert(mode == IRIS_SERVICE_SERVER);
1378         service_socket = socket_;
1379         socketSet.addSocket(service_socket);
1380         adapter.initServiceServer(service_socket, &socketSet, &receivingInterface, verbose);
1381     }
1382
1383 private:
1385     std::string getInstName(InstanceId instId)
1386     {
1387         // IrisLogger will generate a default name for unknown instances (empty string).
1388         return instId < instIdToInstName.size() ? instIdToInstName[instId] : std::string();
1389     }
1390
1392     void setInstName(InstanceId instId, const std::string& instName)
1393     {
1394         // Ignore ridiculously high instIds (prigramming errors).
1395         if (instId >= IrisMaxTotalInstances)
1396         {
1397             return;
1398         }
1399
1400         if (instId >= instIdToInstName.size())
1401         {
1402             instIdToInstName.resize(instId + 1, "");
1403         }
1404
1405         instIdToInstName[instId] = instName;
1406     }
1407
1408     // --- Private data. ---
1409
1411     IrisLogger log;
1412
1414     IrisInstance irisInstance;
1415
1417     std::string irisInstanceInstName;
1418
1420     GlobalInstanceSendingInterface globalInstanceSendingInterface{this};
1421
1423     ReceivingInterface receivingInterface{log, this};
1424
1426     impl::IrisTcpSocket sock{log, 0};
1427
1429     impl::IrisTcpSocket* service_socket{nullptr};
1430
1432     impl::IrisTcpSocketSet socketSet{log, 0};
1433
1435     std::vector<IrisInterface*> instIdToInterface;
1436
1438     std::vector<std::string> instIdToInstName;
1439
1441     impl::IrisChannelRegistry channel_registry{log};
1442
1444     IrisInterface* sendingInterface{nullptr};
```

```
1445
1448     uint32_t nextInstIdForRegisterInstanceCall{0};
1449
1451     struct OngoingInstanceRegistryCallEntry
1452     {
1453         OngoingInstanceRegistryCallEntry()
1454         {
1455         }
1456
1457         OngoingInstanceRegistryCallEntry(const std::string& method_, uint64_t id_ = IRIS_UINT64_MAX,
1458                                          uint64_t channelId_ = IRIS_UINT64_MAX)
1459             : method(method_)
1460             , id(id_)
1461             , channelId(channelId_)
1462         {
1463         }
1464
1465         std::string method;                  // instanceRegistry_registerInstance,
       instanceRegistry_unregisterInstance or instanceRegistry_getList().
1466         uint64_t id{IRIS_UINT64_MAX};        // For instanceRegistry_registerInstance(): Original
       request id. For instanceRegistry_unregisterInstance(): params.aInstId.
1467         uint64_t channelId{IRIS_UINT64_MAX}; // For instanceRegistry_registerInstance() only:
       params.channelId.
1468     };
1469
1473     typedef std::map<uint64_t, OngoingInstanceRegistryCallEntry> OngoingInstanceRegistryCallMap;
1474
1475     OngoingInstanceRegistryCallMap ongoingInstanceRegistryCalls;
1476
1478     std::mutex ongoingInstanceRegistryCallsMutex;
1479
1481     unsigned verbose{0};
1482
1484     unsigned irisMessageLogLevel{0};
1485
1487     impl::IrisRpcAdapterTcp adapter{log};
1488
1490     std::unique_ptr<impl::IrisProcessEventsThread> socket_thread{nullptr};
1491
1493     Mode mode;
1494
1496     std::string component_name;
1497
1499     std::unique_ptr<impl::IrisPlugin<IrisClient» plugin{nullptr};
1500
1502     std::string connectionStr{"(not connected)"};
1503
1506     uint64_t sleepOnDestructionMs{};
1507
1508 #ifndef _WIN32
1510     pid_t childPid{};
1511 #endif
1512
1514     int lastExitStatus{-1};
1515 };
1516
1517 NAMESPACE_IRIS_END
1518
1519 #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

Copyright (C) 2020-2022 Arm Limited. All rights reserved.

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     struct Option
61     {
62         // Public interface:
63
68         Option& setList(char sep = ',') { listSeparator = sep; return *this; }
69
70     private:
71         // Meta info:
72
74         char shortOption{};
75
78         std::string longOption;
79
81         std::string help;
82
86         std::string formalArgumentName;
87
89         std::string defaultValue;
90
92         char listSeparator{};
93
95         bool hasFormalArgument() const { return !formalArgumentName.empty(); }
96
97         // Actual values from command line:
```

```
98
102         std::string value;
103
105         bool isSpecified{};
106
108         void setValue(const std::string& v);
109
111         void unsetValue();
112
113         friend class IrisCommandLineParser;
114     };
115
117     IrisCommandLineParser(const std::string& programName_, const std::string& usageHeader_, const
      std::string& versionStr_);
118
126     Option& addOption(char shortOption, const std::string& longOption, const std::string& help, const
      std::string& formalArgumentName = std::string(), const std::string& defaultValue = std::string());
127
150     int parseCommandLine(int argc, const char** argv);
151     int parseCommandLine(int argc, char** argv) { return parseCommandLine(argc, const_cast<const
      char**>(argv)); }
152
155     void noNonOptionArguments();
156
160     void pleaseSpecifyOneOf(const std::vector<std::string>& options, const std::vector<std::string>&
      formalNonOptionArguments = std::vector<std::string>());
161
163     std::string getStr(const std::string& longOption) const;
164
167     int64_t getInt(const std::string& longOption) const;
168
171     uint64_t getUint(const std::string& longOption) const;
172
175     double getDbl(const std::string& longOption) const;
176
178     uint64_t getSwitch(const std::string& longOption) const;
179
181     bool operator()(const std::string& longOption) const { return getSwitch(longOption) > 0; }
182
184     std::vector<std::string> getList(const std::string& longOption) const;
185
189     std::map<std::string,std::string> getMap(const std::string& longOption) const;
190
194     bool isSpecified(const std::string& longOption) const;
195
197     const std::vector<std::string>& getNonOptionArguments() const;
198
202     void clear();
203
208     int printMessage(const std::string& message, int error = 0, bool exit = false) const;
209
211     int printError(const std::string& message) const;
212
216     int printErrorAndExit(const std::string& message) const;
217
221     int printErrorAndExit(const IrisErrorException& e) const { return printErrorAndExit(e.errorMessage()
      + "\n"); }
222
226     int printErrorAndExit(const std::exception& e) const;
227
239     void setMessageFunc(const std::function<int(const std::string& message, int error, bool exit)>&
      messageFunc);
240
244     static int defaultMessageFunc(const std::string& message, int error, bool exit);
245
249     std::string getHelpMessage() const;
250
254     void setValue(const std::string& longOption, const std::string& value, bool append = false);
255
258     void unsetValue(const std::string& longOption);
259
261     void setProgramName(const std::string& programName_, bool append = false);
262
263 private:
266     Option& getOption(const std::string& longOption);
267
269     const Option& getOption(const std::string& longOption) const;
270
272     std::string programName;
273
275     std::string usageHeader;
276
278     std::string versionStr;
279
281     std::vector<std::string> optionList;
282
285     std::map<std::string, Option> options;
```

```
286
288     std::vector<std::string> nonOptionArguments;
289
291     std::function<int(const std::string& message, int error, bool exit)> messageFunc;
292 };
293
294 NAMESPACE_IRIS_END
295
296 #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.

Copyright

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

Copyright ARM Limited 2014-2019 All Rights Reserved.

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& connection_info =
      "");
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) 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         irisProxyInterface = irisProxyInterface_ ? irisProxyInterface_ : &defaultIrisProxyInterface;
88     }
89
90 public:
91     // IrisInterface implementation.
92
94     virtual void  irisHandleMessage(const uint64_t* message) override;
95
96     // Set log level for logging messages.
97     void setLogLevel(unsigned level);
98
99 private:
100     // --- Functions implemented locally in the global instance (registered in the functionDecoder). ---
101
103     void impl_instanceRegistry_registerInstance(IrisReceivedRequest& request);
104
106     void impl_instanceRegistry_unregisterInstance(IrisReceivedRequest& request);
107
109     void impl_instanceRegistry_getList(IrisReceivedRequest& request);
110
112     void impl_instanceRegistry_getInstanceInfoByInstId(IrisReceivedRequest& request);
113
115     void impl_instanceRegistry_getInstanceInfoByName(IrisReceivedRequest& request);
116
118     void impl_perInstanceExecution_setStateAll(IrisReceivedRequest& request);
119
121     void impl_perInstanceExecution_getStateAll(IrisReceivedRequest& request);
122
```

```
124     void impl_tcpServer_start(IrisReceivedRequest& request);
125
127     void impl_tcpServer_stop(IrisReceivedRequest& request);
128
130     void impl_tcpServer_getPort(IrisReceivedRequest& request);
131
133     void impl_plugin_load(IrisReceivedRequest& request);
134
136     void impl_service_connect(IrisReceivedRequest& request);
137
139     void impl_service_disconnect(IrisReceivedRequest& request);
140
141     // --- Private helpers ---
142
144     struct InstanceRegistryEntry
145     {
146         std::string    instName;
147         uint64_t       channelId{IRIS_UINT64_MAX}; // If this is IRIS_UINT64_MAX this means this entry
    is unused.
148         IrisInterface* iris_interface{nullptr};
149         std::string     connection_info;
150
151         bool empty() const
152         {
153             return channelId == IRIS_UINT64_MAX;
154         }
155
157         void clear()
158         {
159             instName        = "";
160             channelId       = IRIS_UINT64_MAX;
161             iris_interface  = nullptr;
162             connection_info = "";
163
164             assert(empty());
165         }
166     };
167
169     InstanceId registerInstance(std::string&   instName,
170                                 uint64_t       channel_id,
171                                 bool           uniquify,
172                                 IrisInterface* iris_interface);
173
175     void unregisterInstanceAndGenerateEvent(InstanceRegistryEntry*  entry,
176                                             InstanceId              aInstId,
177                                             uint64_t                time,
178                                             std::list<IrisRequest>& deferred_event_requests);
179
181     const InstanceRegistryEntry* findInstanceRegistryEntry(InstanceId instId) const
182     {
183         if (instId >= InstanceId(instanceRegistry.size()))
184             return nullptr;
185
186         if (instanceRegistry[instId].empty())
187             return nullptr;
188
189         return &instanceRegistry[instId];
190     }
191
195     InstanceId addNewInstance(const std::string& instName,
196                               uint64_t           channelId,
197                               IrisInterface*     iris_interface);
198
199     // Stop the Iris Server (if running)
200     void stopServer();
201
202     // stop the Iris Client (if running)
203     void stopClient();
204
205     void loadPlugin(const std::string& plugin_path);
206
208     IrisErrorCode createEventStream(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&);
209
211     uint64_t getTimeForEvents();
212
215     std::string getInstName(InstanceId instId) const;
216
217
218     // --- Private data ---
219
224     class Instance : public IrisInstance
225     {
226     public:
227         Instance()
228             : IrisInstance()
229         {
```

```
230             thisInstanceInfo.instName = "framework.GlobalInstance";
231             thisInstanceInfo.instId   = IrisInstIdGlobalInstance;
232             setProperty("instName", getInstanceName());
233             setProperty("instId", getInstId());
234             // NOTE: This instance does not think it is registered.
235             //       This means it won't unregister itself when it is destroyed but that doesn't matter.
236             //       We will be cleaning up all that state anyway.
237         }
238 
239         IrisInstanceEvent event_handler;
240     } irisInstance;
241 
243     IrisEventRegistry instance_registry_changed_event_registry;
244 
246     IrisEventRegistry shutdown_enter_event_registry;
247 
249     IrisEventRegistry shutdown_leave_event_registry;
250 
253     std::vector<InstanceRegistryEntry> instanceRegistry;
254 
255     //
256     std::mutex instance_registry_mutex;
257 
260     std::vector<InstanceId> freeInstIds;
261 
263     typedef std::map<std::string, uint64_t> InstanceRegistryNameToIdMap;
264 
266     InstanceRegistryNameToIdMap instanceRegistryNameToId;
267 
277     unsigned logMessages;
278 
280     IrisLogger log;
281 
282     // TCP server. This won't start listening until startServer() is called.
283     impl::IrisTcpServer* tcp_server;
284 
286     impl::IrisServiceClient* service_client;
287 
288     // Create and manage communication channels
289     impl::IrisChannelRegistry channel_registry;
290 
291     std::unordered_map<uint64_t, std::string> channel_connection_info;
292     std::mutex                                channel_connection_info_mutex;
293 
294     // --- Load and manage plugins ---
295     using Plugin = impl::IrisPlugin<IrisGlobalInstance>;
296     std::unordered_map<std::string, std::unique_ptr<Plugin» plugins;
297 
298     std::mutex plugins_mutex;
299 
300     std::mutex log_mutex;
301 
306     class DefaultIrisProxyInterface : public IrisProxyInterface
307     {
308     public:
309         virtual void          irisHandleMessageInProxy(IrisInterface* irisInterface, InstanceId instId,
    const uint64_t* message) override;
310         virtual IrisErrorCode processAsyncMessagesInProxy(bool waitForAMessage) override;
311     } defaultIrisProxyInterface;
312 
320     std::atomic<IrisProxyInterface*> irisProxyInterface{&defaultIrisProxyInterface};
321 };
322 
323 NAMESPACE_IRIS_END
324 
325 #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/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

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/IrisInstanceEvent.h"
28
29 #include <cassert>
30 #include <mutex>
31 #include <functional>
32
33 NAMESPACE_IRIS_START
34
65 typedef IrisDelegate<uint64_t, const IrisValueMap&, uint64_t, uint64_t, bool, std::string&>
      EventCallbackDelegate;
66 typedef IrisDelegate<const EventBufferCallbackData&> EventBufferCallbackDelegate;
67
68 class IrisInstantiationContext;
69 class IrisInstanceBuilder;
70
71 class IrisInstance
72 {
73 public:
74 // --- Construction and destruction. ---
75
82 #define irisRegisterFunction(instancePtr, instanceType, functionName, functionInfoJson)
      registerFunction(instancePtr, #functionName, &instanceType::impl_##functionName, functionInfoJson,
      #instanceType)
83
87 #define irisRegisterEventCallback(instancePtr, instanceType, functionName, description)
      registerEventCallback<instanceType, &instanceType::impl_##functionName>(instancePtr, #functionName,
      description, #instanceType)
88
90 #define irisRegisterEventBufferCallback(instancePtr, instanceType, functionName, description)
      registerEventBufferCallback<instanceType, &instanceType::impl_##functionName>(instancePtr,
      #functionName, description, #instanceType)
91
99     static const uint64_t UNIQUIFY = (1 << 0);
100
102     static const uint64_t THROW_ON_ERROR = (1 << 1);
103
105     static const uint64_t DEFAULT_FLAGS = THROW_ON_ERROR;
106
126     IrisInstance(IrisConnectionInterface* connection_interface = nullptr,
127                  const std::string&       instName             = std::string(),
128                  uint64_t                 flags                = DEFAULT_FLAGS);
129
136     IrisInstance(IrisInstantiationContext* context);
137
139     ~IrisInstance();
```

```
140
148     void setConnectionInterface(IrisConnectionInterface* connection_interface);
149
154     void processAsyncRequests();
155
161     IrisInterface* getRemoteIrisInterface()
162     {
163         return remoteIrisInterface;
164     }
165
176     void setThrowOnError(bool throw_on_error)
177     {
178         default_cppAdapter = throw_on_error ? &throw_cppAdapter : &nothrow_cppAdapter;
179     }
180
194     IrisErrorCode registerInstance(const std::string& instName, uint64_t flags = DEFAULT_FLAGS);
195
201     IrisErrorCode unregisterInstance();
202
221     template <class T>
222     void setProperty(const std::string& propertyName, const T& propertyValue)
223     {
224         propertyMap[propertyName].set(propertyValue);
225     }
226
233     const PropertyMap& getPropertyMap() const
234     {
235         return propertyMap;
236     }
237
238     // --- Interface for components. Provide functionality to clients. ---
239
252     template <class T>
253     void registerFunction(T* instance, const std::string& name, void
    (T::*memberFunctionPtr)(IrisReceivedRequest&), const std::string& functionInfoJson, const
    std::string& instanceTypeStr)
254     {
255         functionDecoder.registerFunction(instance, name, memberFunctionPtr, functionInfoJson,
    instanceTypeStr);
256     }
257
261     void unregisterFunction(const std::string& name)
262     {
263         functionDecoder.unregisterFunction(name);
264     }
265
277     template <class T>
278     void registerEventCallback(T* instance, const std::string& name, const std::string& description,
279                                void (T::*memberFunctionPtr)(IrisReceivedRequest&),
280                                const std::string& instanceTypeStr)
281     {
282         std::string funcInfoJson = "{description:'" + description +
283             "',"
284             "args:{"
285             "  instId:{type:'NumberU64', description:'Target instance id.'},"
286             "  esId:{type:'NumberU64', description:'Event stream id.'},"
287             "  fields:{type:'Object', description:'Object which contains the names and values of event
    source fields.'},"
288             "  time:{type:'NumberU64', description:'Simulation time timestamp of the event.'},"
289             "  sInstId:{type:'NumberU64', description:'Source instId: Instance which generated and sent
    this event.'},"
290             "  syncEc:{type:'Boolean', description:'Synchronous callback behaviour.', optional:true},"
291             "},"
292             "retval:{type:'Null'}}";
293         functionDecoder.registerFunction(instance, name, memberFunctionPtr, funcInfoJson,
    instanceTypeStr);
294     }
295
304     void registerEventCallback(EventCallbackDelegate delegate, const std::string& name,
305                                const std::string& description, const std::string& dlgInstanceTypeStr)
306     {
307         eventCallbacks[name] = ECD(delegate);
308         registerEventCallback(this, name, description, &IrisInstance::impl_eventCallback,
    dlgInstanceTypeStr);
309     }
310
319     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, const AttributeValueMap&, uint64_t,
    uint64_t, bool, std::string&)>
320     void registerEventCallback(T* instance, const std::string& name, const std::string& description,
321                                const std::string& dlgInstanceTypeStr)
322     {
323         registerEventCallback(EventCallbackDelegate::make<T, METHOD>(instance),
324                               name, description, dlgInstanceTypeStr);
325     }
326
338     template <class T>
339     void registerEventBufferCallback(T* instance, const std::string& name, const std::string&
```

```
    description,
340                                 void (T::*memberFunctionPtr)(IrisReceivedRequest&),
341                                 const std::string& instanceTypeStr)
342     {
343         std::string funcInfoJson = "{description:'" + description + "',"
344             "args:{"
345             "  instId:{type:'NumberU64', description:'Target instance id.'},"
346             "  sInstId:{type:'NumberU64', description:'Source instId: Instance which generated and sent
    this event buffer data.'},"
347             "  evBufId:{type:'NumberU64', description:'Event buffer id.'},"
348             "  events:{type:'EventData[]', description:'Array of EventData objects which represent the
    individual events in chronological order.'}"
349             "},"
350             "retval:{type:'Null'}}";
351         functionDecoder.registerFunction(instance, name, memberFunctionPtr, funcInfoJson,
    instanceTypeStr);
352     }
353
362     void registerEventBufferCallback(EventBufferCallbackDelegate delegate, const std::string& name,
363                                 const std::string& description, const std::string&
    dlgInstanceTypeStr)
364     {
365         eventBufferCallbacks[name] = EBCD(delegate);
366         registerEventBufferCallback(this, name, description, &IrisInstance::impl_eventBufferCallback,
    dlgInstanceTypeStr);
367     }
368
377     template <typename T, IrisErrorCode (T::*METHOD)(const EventBufferCallbackData& data)>
378     void registerEventBufferCallback(T* instance, const std::string& name, const std::string&
    description,
379                                 const std::string& dlgInstanceTypeStr)
380     {
381         registerEventBufferCallback(EventBufferCallbackDelegate::make<T, METHOD>(instance),
382                                     name, description, dlgInstanceTypeStr);
383     }
384
388     void unregisterEventCallback(const std::string& name);
389
393     void unregisterEventBufferCallback(const std::string& name);
394
405     using EventCallbackFunction = std::function<IrisErrorCode(EventStreamId, const IrisValueMap&,
    uint64_t, InstanceId, bool, std::string&)>;
406
410     void setCallback_IRIS_SIMULATION_TIME_EVENT(EventCallbackFunction f);
411
415     void setCallback_IRIS_SHUTDOWN_LEAVE(EventCallbackFunction f);
416
420     void addCallback_IRIS_INSTANCE_REGISTRY_CHANGED(EventCallbackFunction f);
421
429     void sendResponse(const uint64_t* response)
430     {
431         remoteIrisInterface->irisHandleMessage(response);
432     }
433
434     // --- Interface for clients. Access to other components. ---
435
441     IrisCppAdapter& irisCall() { return *default_cppAdapter; }
442
450     IrisCppAdapter& irisCallNoThrow() { return nothrow_cppAdapter; }
451
466     IrisCppAdapter& irisCallThrow() { return throw_cppAdapter; }
467
479     bool sendRequest(IrisRequest& req)
480     {
481         return irisCall().callAndPerhapsWaitForResponse(req);
482     }
483
484     // --- Misc functionality. ---
485
492     IrisInterface* getLocalIrisInterface() { return functionDecoder.getIrisInterface(); }
493
500     InstanceId getInstId() const { return thisInstanceInfo.instId; }
501
507     void setInstId(InstanceId instId) { thisInstanceInfo.instId = instId;
    cppAdapter_request_manager.setInstId(instId); }
508
518     const std::string& getInstanceName() const { return thisInstanceInfo.instName; }
519
522     bool isRegistered() const { return cppAdapter_request_manager.isRegistered(); }
523
530     IrisInstanceBuilder* getBuilder();
531
532     bool isAdapterInitialized() const { return is_adapter_initialized; }
533
534     void setAdapterInitialized() { is_adapter_initialized = true; }
535
539     void setEventHandler(IrisInstanceEvent* handler);
```

```
540
551     void notifyStateChanged();
552
563     template<class T>
564     void publishCppInterface(const std::string& interfaceName, T *pointer, const std::string&
        jsonDescription)
565     {
566         // Ignore null pointers: instance_getCppInterface...() promises to always return non-null
        pointers.
567         // (If there is no interface, do not publish it.)
568         if (pointer == nullptr)
569             return;
570
571         std::string functionInfoJson =
572             "{"
573             "    \"description\": \"" + jsonDescription + "\n"
574             "If this function is present it always returns a non-null pointer.\n"
575             "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. "
576             "This effectively means that they both must be compiled using the same compiler using the
        same header files. "
577             "The returned pointer is only meaningful if caller and callee run in the same process.\n"
578             "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'.\","
579             "    \"args\": {"
580             "        \"instId\": {"
581             "            \"description\": \"Opaque number uniquely identifying the target instance.\","
582             "            \"type\": \"NumberU64\""
583             "        }"
584             "    },"
585             "    \"errors\": ["
586             "        \"E_unknown_instance_id\""
587             "    ],"
588             "    \"retval\": {"
589             "        \"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.\","
590             "        \"type\": \"CppInterfacePointer\""
591             "    }"
592             "}";
593         registerFunction(this, "instance_getCppInterface" + interfaceName,
        &IrisInstance::impl_instance_getCppInterface, functionInfoJson, "IrisInstance");
594         cppInterfaceRegistry[interfaceName].set(pointer);
595     }
596
605     void unpublishCppInterface(const std::string& interfaceName)
606     {
607         unregisterFunction("instance_getCppInterface" + interfaceName);
608         cppInterfaceRegistry.erase(interfaceName);
609     }
610
611     // --- Blocking simulation time functions ---
612
620     void simulationTimeRun();
621
627     void simulationTimeStop();
628
634     void simulationTimeRunUntilStop(double timeoutInSeconds = 0.0);
635
649     bool simulationTimeWaitForStop(double timeoutInSeconds = 0.0);
650
659     bool simulationTimeIsRunning();
660
673     void simulationTimeDisableEvents();
674
681     void setPendingSyncStepResponse(RequestId requestId, EventBufferId evBufId);
682
690     bool isValidEvBufId(EventBufferId evBufId) const;
691
735     std::vector<EventStreamInfo> findEventSourcesAndFields(const std::string& spec, InstanceId
        defaultInstId = IRIS_UINT64_MAX);
736     void findEventSourcesAndFields(const std::string& spec, std::vector<EventStreamInfo>&
        eventStreamInfosOut, InstanceId defaultInstId = IRIS_UINT64_MAX);
737
738
783     void enableEvent(const std::string& eventSpec, std::function<void (const EventStreamInfo&
        eventStreamInfo, IrisReceivedRequest& request)> callback);
784
797     void enableEvent(const std::string& eventSpec, std::function<void ()> callback);
798
817     void disableEvent(const std::string& eventSpec);
818
826     std::vector<InstanceInfo> findInstanceInfos(const std::string& instancePathFilter = "all");
827
834     std::vector<EventSourceInfo> findEventSources(const std::string& instancePathFilter = "all");
835
```

```
843     const InstanceInfo& getInstanceInfo(InstanceId instId);
844
861     InstanceInfo getInstanceInfo(const std::string& instancePathFilter);
862
873     const std::vector<InstanceInfo>& getInstanceList();
874
884     std::string getInstanceName(InstanceId instId);
885
895     InstanceId getInstanceId(const std::string& instName);
896
906     ResourceId getResourceId(InstanceId instId, const std::string& resourceSpec);
907
931     uint64_t resourceRead(InstanceId instId, const std::string& resourceSpec);
932
940     uint64_t resourceReadCrn(InstanceId instId, uint64_t canonicalRegisterNumber)
941     {
942         return resourceRead(instId, "crn:" + std::to_string(canonicalRegisterNumber));
943     }
944
954     std::string resourceReadStr(InstanceId instId, const std::string& resourceSpec);
955
963     void resourceWrite(InstanceId instId, const std::string& resourceSpec, uint64_t value);
964
970     void resourceWriteCrn(InstanceId instId, uint64_t canonicalRegisterNumber, uint64_t value)
971     {
972         resourceWrite(instId, "crn:" + std::to_string(canonicalRegisterNumber), value);
973     }
974
983     void resourceWriteStr(InstanceId instId, const std::string& resourceSpec, const std::string& value);
984
988     const std::vector<ResourceGroupInfo>& getResourceGroups(InstanceId instId);
989
993     const ResourceInfo& getResourceInfo(InstanceId instId, ResourceId resourceId);
994
998     const ResourceInfo& getResourceInfo(InstanceId instId, const std::string& resourceSpec);
999
1003     const std::vector<ResourceInfo>& getResourceInfos(InstanceId instId);
1004
1008     MemorySpaceId getMemorySpaceId(InstanceId instId, uint64_t canonicalMsn);
1009
1016     MemorySpaceId getMemorySpaceId(InstanceId instId, const std::string& name);
1017
1021     const MemorySpaceInfo& getMemorySpaceInfo(InstanceId instId, uint64_t canonicalMsn);
1022
1029     const MemorySpaceInfo& getMemorySpaceInfo(InstanceId instId, const std::string& name);
1030
1034     const std::vector<MemorySpaceInfo>& getMemorySpaceInfos(InstanceId instId);
1035
1039     void clearCachedMetaInfo();
1040
1041 private:
1042     void init(IrisConnectionInterface* connection_interface_ = nullptr,
1043               const std::string&       instName              = std::string(),
1044               uint64_t                 flags                 = DEFAULT_FLAGS);
1045
1047     struct InstanceMetaInfo
1048     {
1051         std::map<std::string,ResourceId> resourceSpecToResourceIdAll;
1052
1056         std::map<std::string,ResourceId> resourceSpecToResourceIdUsed;
1057
1059         std::vector<ResourceGroupInfo> groupInfos;
1060
1062         std::vector<ResourceInfo> resourceInfos;
1063
1065         std::map<ResourceId,uint64_t> resourceIdToIndex;
1066
1068         std::vector<MemorySpaceInfo> memorySpaceInfos;
1069     };
1070
1074     InstanceMetaInfo& getInstanceMetaInfo(InstanceId instId);
1075
1079     IrisInstance::InstanceMetaInfo& getResourceMetaInfo(InstanceId instId);
1080
1084     IrisInstance::InstanceMetaInfo& getMemoryMetaInfo(InstanceId instId);
1085
1099     void expandWildcardsInEventStreamInfos(std::vector<EventStreamInfo>& eventStreamInfosInOut,
        InstanceId defaultInstId = IRIS_UINT64_MAX, const std::string& instancePathFilter = "all");
1100
1102     void enableSimulationTimeEvents();
1103
1105     void enableShutdownLeaveEvents();
1106
1108     void enableInstanceRegistryChangedEvent();
1109
1111     void simulationTimeWaitForRunning();
1112
```

```
1114     void simulationTimeClearGotRunning();
1115
1119     std::string lookupInstanceNameLocal(InstanceId instId);
1120
1121     // --- Iris function implementations ---
1122     void impl_instance_getProperties(IrisReceivedRequest& request);
1123
1124     void impl_instance_ping(IrisReceivedRequest& request);
1125
1126     void impl_instance_ping2(IrisReceivedRequest& request);
1127
1128     void impl_instance_getCppInterface(IrisReceivedRequest& request);
1129
1131     void impl_eventCallback(IrisReceivedRequest& request);
1132
1134     void impl_eventBufferCallback(IrisReceivedRequest& request);
1135
1137     void impl_enableEventCallback(IrisReceivedRequest &request);
1138
1140     IrisErrorCode impl_ec_IrisInstance_IRIS_SIMULATION_TIME_EVENT(EventStreamId esId, const
      IrisValueMap& fields, uint64_t time,
1141                                                                  InstanceId sInstId, bool syncEc,
      std::string& errorMessageOut);
1142
1144     IrisErrorCode impl_ec_IrisInstance_IRIS_SHUTDOWN_LEAVE(EventStreamId esId, const IrisValueMap&
      fields, uint64_t time,
1145                                                           InstanceId sInstId, bool syncEc,
      std::string& errorMessageOut);
1146
1148     IrisErrorCode impl_ec_IrisInstance_IRIS_INSTANCE_REGISTRY_CHANGED(EventStreamId esId, const
      IrisValueMap& fields, uint64_t time,
1149                                                                      InstanceId sInstId, bool syncEc,
      std::string& errorMessageOut);
1150
1151     // --- Iris specific data and state ---
1152
1154     IrisFunctionDecoder functionDecoder{log, this};
1155
1157     IrisCppAdapter::RequestManager cppAdapter_request_manager{log};
1158
1160     IrisCppAdapter throw_cppAdapter{&cppAdapter_request_manager, /*throw_on_error=*/true};
1161
1163     IrisCppAdapter nothrow_cppAdapter{&cppAdapter_request_manager, /*throw_on_error=*/false};
1164
1166     IrisCppAdapter* default_cppAdapter{&throw_cppAdapter};
1167
1171     IrisConnectionInterface* connection_interface{nullptr};
1172
1175     IrisInterface* remoteIrisInterface{nullptr};
1176
1177 protected:
1179     InstanceInfo thisInstanceInfo{};
1180
1181 private:
1183     bool instance_getProperties_called{false};
1184
1185     bool registered{false};
1186
1187     bool is_adapter_initialized{false};
1188
1189     uint64_t channelId{IRIS_UINT64_MAX};
1190
1192     IrisLogger log;
1193
1194     // --- Instance specific data and state ---
1195
1197     PropertyMap propertyMap{};
1198
1200     struct ECD
1201     {
1202         // Work around symbol length limits in Visual Studio (warning C4503)
1203         EventCallbackDelegate dlg;
1204         ECD() {}
1205         ECD(EventCallbackDelegate dlg_)
1206             : dlg(dlg_)
1207         {
1208         }
1209     };
1210     typedef std::map<std::string, ECD> EventCallbackMap;
1211     EventCallbackMap                   eventCallbacks{};
1212
1214     struct EBCD
1215     {
1216         // Work around symbol length limits in Visual Studio (warning C4503)
1217         EventBufferCallbackDelegate dlg;
1218         EBCD() {}
1219         EBCD(EventBufferCallbackDelegate dlg_)
```

```
1220            : dlg(dlg_)
1221        {
1222        }
1223    };
1224    typedef std::map<std::string, EBCD> EventBufferCallbackMap;
1225    EventBufferCallbackMap              eventBufferCallbacks{};
1226
1228    struct EnableEventCallbackInfo
1229    {
1230        EnableEventCallbackInfo() = default;
1231        EnableEventCallbackInfo(const EventStreamInfo& eventStreamInfo_, std::function<void (const
      EventStreamInfo& eventStreamInfo, IrisReceivedRequest& request)> callback_):
1232            eventStreamInfo(eventStreamInfo_),
1233            callback(callback_)
1234        {
1235        }
1236
1237        EventStreamInfo eventStreamInfo;
1238        std::function<void (const EventStreamInfo& eventStreamInfo, IrisReceivedRequest& request)>
      callback;
1239    };
1240    typedef std::map<std::string, EnableEventCallbackInfo> EnableEventCallbackMap;
1241    EnableEventCallbackMap enableEventCallbacks;
1242
1244    IrisInstanceBuilder* builder{nullptr};
1245
1247    IrisInstanceEvent *irisInstanceEvent{};
1248
1252    typedef std::map<std::string, CppInterfacePointer> CppInterfaceRegistryMap;
1253    CppInterfaceRegistryMap cppInterfaceRegistry{};
1254
1256    bool simulationTimeIsRunning_{};
1257
1259    bool simulationTimeGotRunningTrue{};
1260
1262    bool simulationTimeGotRunningFalse{};
1263
1265    std::mutex simulationTimeIsRunningMutex;
1266
1268    std::condition_variable simulationTimeIsRunningChanged;
1269
1271    EventStreamId simulationTimeEsId = IRIS_UINT64_MAX;
1272
1274    EventStreamId shutdownLeaveEsId = IRIS_UINT64_MAX;
1275
1277    EventStreamId instanceRegistryChangedEsId = IRIS_UINT64_MAX;
1278
1280    EventCallbackFunction simulationTimeCallbackFunction;
1281
1283    EventCallbackFunction shutdownLeaveCallbackFunction;
1284
1285    // List of callback functions for IRIS_INSTANCE_REGISTRY_CHANGED.
1286    std::vector<EventCallbackFunction> instanceRegistryChangedFunctions;
1287
1289    struct PendingSyncStepResponse
1290    {
1292        void set(RequestId requestId_, EventBufferId evBufId_)
1293        {
1294            requestId = requestId_;
1295            evBufId = evBufId_;
1296        }
1297
1299        bool isPending() const
1300        {
1301            return requestId != IRIS_UINT64_MAX;
1302        }
1303
1305        void clear()
1306        {
1307            requestId = IRIS_UINT64_MAX;
1308            evBufId = 0;
1309        }
1310
1313        RequestId requestId{IRIS_UINT64_MAX};
1314
1316        EventBufferId evBufId{};
1317    };
1318
1320    PendingSyncStepResponse pendingSyncStepResponse;
1321
1323    std::vector<InstanceInfo> instanceInfos;
1324
1327    std::vector<uint64_t> instIdToIndex;
1328
1330    std::map<InstanceId,InstanceMetaInfo> instIdToMetaInfo;
1331
1334    bool globalInstanceHasNotifyStateChangedChecked{};
```

```
1335
1338     bool globalInstanceHasNotifyStateChanged{};
1339 };
1340
1341
1342 NAMESPACE_IRIS_END
1343
1344 #endif // #ifndef ARM_INCLUDE_IrisInstance_h
1345
1346 // Convenience #include.
1347 // (IrisInstanceBuilder needs the complete type of IrisInstance.)
1348 #include "iris/IrisInstanceBuilder.h"
1349
```

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>
```

Classes

- struct iris::BreakpointHitInfo
- 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.
- typedef IrisDelegate< const BreakpointHitInfo & > iris::HandleBreakpointHitDelegate

  Handle a breakpoint hit corresponding to the given information.

9.17.1 Detailed Description

Breakpoint add-on to IrisInstance.

Copyright

Copyright (C) 2016-2020 Arm Limited. All rights reserved.

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.

9.17.2.3 HandleBreakpointHitDelegate

```
typedef IrisDelegate<const BreakpointHitInfo&> iris::HandleBreakpointHitDelegate
```

Handle a breakpoint hit corresponding to the given information.

```
IrisErrorCode handleBreakpointHit(const BreakpointHitInfo &bptInfo)
```

The breakpoint is guaranteed to exist and to be valid.

Error: Return E_∗ error code if there is some error in handling 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
22 NAMESPACE_IRIS_START
23
24 class IrisInstance;
25 class IrisInstanceEvent;
26 class IrisEventRegistry;
27 class IrisReceivedRequest;
28
29 class EventStream;
30 struct EventSourceInfo;
31
32 struct BreakpointHitInfo
33 {
34     //Required for all breakpoint types
35     const BreakpointInfo& bptInfo;
36
37     //Register and memory breakpoint
38     const std::vector<uint64_t>& accessData;
39     bool isReadAccess;
40 };
41
55 typedef IrisDelegate<BreakpointInfo&> BreakpointSetDelegate;
56
67 typedef IrisDelegate<const BreakpointInfo&> BreakpointDeleteDelegate;
68
79 typedef IrisDelegate<const BreakpointHitInfo&> HandleBreakpointHitDelegate;
80
101 class IrisInstanceBreakpoint
102 {
103
104 public:
105     // --- Construction and destruction. ---
106     IrisInstanceBreakpoint(IrisInstance* irisInstance = nullptr);
107     ~IrisInstanceBreakpoint();
108
116     void attachTo(IrisInstance* irisInstance);
```

```
117
123     void setBreakpointSetDelegate(BreakpointSetDelegate delegate);
124
130     void setBreakpointDeleteDelegate(BreakpointDeleteDelegate delegate);
131
137     void setHandleBreakpointHitDelegate(HandleBreakpointHitDelegate delegate);
138
144     void setEventHandler(IrisInstanceEvent* handler);
145
157     void notifyBreakpointHit(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId);
158
176     void notifyBreakpointHitData(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
177                                  uint64_t accessAddr, uint64_t accessSize,
178                                  const std::string& accessRw, const std::vector<uint64_t>& data);
179
195     void notifyBreakpointHitRegister(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
196                                      const std::string& accessRw, const std::vector<uint64_t>& data);
197
205     const BreakpointInfo* getBreakpointInfo(BreakpointId bptId) const;
206
216     void addCondition(const std::string& name, const std::string& type, const std::string& description,
217                       const std::vector<std::string> bpt_types = std::vector<std::string>());
218
225     void handleBreakpointHit(const BreakpointHitInfo& bptHitInfo);
226
227 private:
228     void impl_breakpoint_set(IrisReceivedRequest& request);
229
230     void impl_breakpoint_delete(IrisReceivedRequest& request);
231
232     void impl_breakpoint_getList(IrisReceivedRequest& request);
233
234     void impl_breakpoint_getAdditionalConditions(IrisReceivedRequest& request);
235
236     bool validateInterceptionParameters(IrisReceivedRequest& request, const InterceptionParams&
    interceptionParams);
237
240     bool beginBreakpointHit(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId);
241
243     IrisErrorCode createEventStream(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&);
244
246     IrisErrorCode deleteBreakpoint(BreakpointId bpt);
247
248     void register_ec_IRIS_INSTANCE_REGISTRY_CHANGED();
249     IrisErrorCode ec_IRIS_INSTANCE_REGISTRY_CHANGED(EventStreamId esId, const IrisValueMap& fields,
    uint64_t time,
250                                                     InstanceId sInstId, bool syncEc, std::string&
    errorMessageOut);
251
253
255     IrisInstance* irisInstance;
256
258     IrisEventRegistry* breakpoint_hit_registry;
259
262     std::vector<BreakpointInfo> bptInfos;
263
266     std::vector<uint64_t> freeBptIds;
267
269     std::map<uint64_t, BreakpointAction> bptActions;
270
272     std::vector<BreakpointConditionInfo> additional_conditions;
273
275     BreakpointSetDelegate bptSetDelegate;
276
278     BreakpointDeleteDelegate bptDeleteDelegate;
279
281     HandleBreakpointHitDelegate handleBreakpointHitDelegate;
282
284     IrisLogger log;
285
290     bool instance_registry_changed_registered{};
291 };
292
293 NAMESPACE_IRIS_END
294
295 #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>
```

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.
- 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
30 NAMESPACE_IRIS_START
31
32 class IrisRegisterEventEmitterBase;
33
39 class IrisInstanceBuilder
40 {
41 private:
48     template <typename T, T* (IrisInstanceBuilder::*INIT_METHOD)()>
49     class LazyAddOn
50     {
51     private:
52         IrisInstanceBuilder* parent;
53         T*                   add_on;
54
55     public:
56         LazyAddOn(IrisInstanceBuilder* parent_)
57             : parent(parent_)
58             , add_on(nullptr)
59         {
60         }
61
62         ~LazyAddOn()
63         {
64             delete add_on;
65         }
66
67         T* operator->()
68         {
69             if (add_on == nullptr)
70             {
71                 init();
72             }
73
74             return add_on;
75         }
76
77         operator T*()
78         {
79             if (add_on == nullptr)
80             {
81                 init();
82             }
83
84             return add_on;
85         }
86
87         T* getPtr()
88         {
89             return add_on;
90         }
91
92         void init()
93         {
94             assert(add_on == nullptr);
95             add_on = (parent->*INIT_METHOD)();
96         }
97     };
98     IrisInstance* iris_instance;
99 #define INTERNAL_LAZY(addon) \
100     addon* init##addon();    \
101     LazyAddOn<addon, &IrisInstanceBuilder::init##addon>
102     INTERNAL_LAZY(IrisInstanceResource)
103     inst_resource;
104     INTERNAL_LAZY(IrisInstanceEvent)
105     inst_event;
106     INTERNAL_LAZY(IrisInstanceBreakpoint)
107     inst_breakpoint;
108     INTERNAL_LAZY(IrisInstanceMemory)
```

```
109     inst_memory;
110     INTERNAL_LAZY(IrisInstanceImage)
111     inst_image;
112     INTERNAL_LAZY(IrisInstanceImage_Callback)
113     inst_image_cb;
114     INTERNAL_LAZY(IrisInstanceStep)
115     inst_step;
116     INTERNAL_LAZY(IrisInstancePerInstanceExecution)
117     inst_per_inst_exec;
118     INTERNAL_LAZY(IrisInstanceTable)
119     inst_table;
120     INTERNAL_LAZY(IrisInstanceDisassembler)
121     inst_disass;
122     INTERNAL_LAZY(IrisInstanceDebuggableState)
123     inst_dbg_state;
124     INTERNAL_LAZY(IrisInstanceSemihosting)
125     inst_semihost;
126     INTERNAL_LAZY(IrisInstanceCheckpoint)
127     inst_checkpoint;
128 #undef INTERNAL_LAZY
129
131
132
134     ResourceReadDelegate  default_reg_read_delegate;
135     ResourceWriteDelegate default_reg_write_delegate;
136
139     bool canonicalRnSchemeIsAlreadySet{};
141
143
144     struct RegisterEventInfo
145     {
146         IrisInstanceEvent::EventSourceInfoAndDelegate event_info;
147
148         typedef std::vector<uint64_t> RscIdList;
149         RscIdList                     rscId_list;
150         IrisRegisterEventEmitterBase* event_emitter;
151
152         RegisterEventInfo()
153             : event_emitter(nullptr)
154         {
155         }
156     };
157
158     std::vector<RegisterEventInfo*> register_read_event_info_list;
159     std::vector<RegisterEventInfo*> register_update_event_info_list;
160
161     RegisterEventInfo* active_register_read_event_info{};
162     RegisterEventInfo* active_register_update_event_info{};
163
164     RegisterEventInfo* find_register_event(const std::vector<RegisterEventInfo*>&
    register_event_info_list,
165                                            const std::string&                     name);
166
167     RegisterEventInfo* initRegisterReadEventInfo(const std::string& name);
168     RegisterEventInfo* initRegisterUpdateEventInfo(const std::string& name);
169
172     void finalizeRegisterEvent(RegisterEventInfo* event_info, bool is_read);
174     void associateRegisterWithTraceEvents(ResourceId rscId);
176
178
179     IrisErrorCode setBreakpoint(BreakpointInfo& info);
180     IrisErrorCode deleteBreakpoint(const BreakpointInfo& info);
181
182     struct RegisterEventEmitterPair
183     {
184         IrisRegisterEventEmitterBase* read;
185         IrisRegisterEventEmitterBase* update;
186
187         RegisterEventEmitterPair()
188             : read(nullptr)
189             , update(nullptr)
190         {
191         }
192     };
193     typedef std::map<uint64_t, RegisterEventEmitterPair> RscIdEventEmitterMap;
194     RscIdEventEmitterMap                                 register_event_emitter_map;
195
196     BreakpointSetDelegate    user_setBreakpoint;
197     BreakpointDeleteDelegate user_deleteBreakpoint;
199
200 public:
206     IrisInstanceBuilder(IrisInstance* iris_instance);
207
208     /* No destructor: IrisInstanceBuilder objects live as long as the instance
209      * they belong to. Do not key anything to the destructor.
210      */
211
```

```
218 #define INTERNAL_RESOURCE_BUILDER_MIXIN(TYPE)
                \
219                                                                                 \
220                                                 \
221                     \
222     TYPE& setName(const std::string& name)
                \
223     {
                \
224         info->resourceInfo.name = name;
                \
225         return *this;
                \
226     }
                \
227                                                                                 \
228                                                 \
229                     \
230     TYPE& setCname(const std::string& cname)
                \
231     {
                \
232         info->resourceInfo.cname = cname;
                \
233         return *this;
                \
234     }
                \
235                                                                                 \
236                                 \
237                     \
238     TYPE& setDescription(const std::string& description)
                \
239     {
                \
240         info->resourceInfo.description = description;
                \
241         return *this;
                \
242     }
                \
243                                                 \
244     /* [[deprecated("Inconsistently named function. Use setDescription() instead.")]] */
                \
245     TYPE& setDescr(const std::string& description)
                \
246     {
                \
247         return setDescription(description);
                \
248     }
                \
249                                                                                 \
250                                                 \
251                     \
252     TYPE& setFormat(const std::string& format)
                \
253     {
                \
254         info->resourceInfo.format = format;
                \
255         return *this;
                \
256     }
                \
257                                                                                 \
258                                     \
259                     \
260     TYPE& setBitWidth(uint64_t bitWidth)
                \
261     {
                \
262         info->resourceInfo.bitWidth = bitWidth;
                \
263         return *this;
                \
264     }
                \
265                                                                                 \
266                                                 \
267                     \
268     TYPE& setType(const std::string& type)
                \
269     {
                \
270         info->resourceInfo.type = type;
                \
```

```
271         return *this;
                \
272     }
                \
273                                                                                 \
274                                                 \
275                     \
276     TYPE& setRwMode(const std::string& rwMode)
                \
277     {
                \
278         info->resourceInfo.rwMode = rwMode;
                \
279         return *this;
                \
280     }
                \
281                                                                             \
282                                             \
283                     \
284     TYPE& setSubRscId(uint64_t subRscId)
                \
285     {
                \
286         info->resourceInfo.subRscId = subRscId;
                \
287         return *this;
                \
288     }
                \
289                                             \
290                                             \
291                             \
292                                                     \
293                                                 \
294                     \
295     TYPE& addEnum(const std::string& symbol, const IrisValue& value, const std::string& description =
    std::string())      \
296     {
                \
297         info->resourceInfo.enums.push_back(EnumElementInfo(value, symbol, description));
                 \
298         return *this;
                \
299     }
                \
300                                             \
301                                             \
302                 \
303                                                 \
304                     \
305     TYPE& addStringEnum(const std::string& stringValue, const std::string& description = std::string())
                  \
306     {
                \
307         info->resourceInfo.enums.push_back(EnumElementInfo(IrisValue(stringValue), std::string(),
    description));          \
308         return *this;
                \
309     }
                \
310                                                     \
311                                                             \
312                     \
313     TYPE& setTag(const std::string& tag)
                \
314     {
                \
315         info->resourceInfo.tags[tag] = IrisValue(true);
                \
316         return *this;
                \
317     }
            \
318                                                                 \
319                                                                \
320                                                                \
321                     \
322     TYPE& setTag(const std::string& tag, const IrisValue& value)
                \
323     {
                \
324         info->resourceInfo.tags[tag] = value;
                \
325         return *this;
                \
326     }
```

```
                \
327                                                             \
328                                                         \
329                                                     \
330                                                         \
331                 \
332     TYPE& setReadDelegate(ResourceReadDelegate readDelegate)
                \
333     {
                \
334         info->readDelegate = readDelegate;
                \
335         return *this;
                \
336     }
                \
337                                                             \
338                                                         \
339                                                     \
340                                                         \
341                 \
342     TYPE& setWriteDelegate(ResourceWriteDelegate writeDelegate)
                \
343     {
                \
344         info->writeDelegate = writeDelegate;
                \
345         return *this;
                \
346     }
                \
347                                                             \
348                                             \
349                                                     \
350                                                 \
351 \
352                                                 \
353                                         \
354                 \
355     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, ResourceReadResult&)>
                \
356     TYPE& setReadDelegate(T* instance)
                \
357     {
                \
358         return setReadDelegate(ResourceReadDelegate::make<T, METHOD>(instance));
                \
359     }
                \
360                                                             \
361                                                              \
362                                                         \
363                                                     \
364                                                         \
365                 \
366     template <IrisErrorCode (*FUNC)(const ResourceInfo&, ResourceReadResult&)>
                \
367     TYPE& setReadDelegate()
                \
368     {
                \
369         return setReadDelegate(ResourceReadDelegate::make<FUNC>());
                \
370     }
                \
371                                                             \
372                                             \
373                                                     \
374                                                 \
375 \
376                                                 \
377                                         \
378                 \
379     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, const ResourceWriteValue&)>
                \
380     TYPE& setWriteDelegate(T* instance)
                \
381     {
                \
382         return setWriteDelegate(ResourceWriteDelegate::make<T, METHOD>(instance));
                \
383     }
                \
384                                                             \
385                                                              \
386                                                         \
387                                                     \
```

```
388                                                 \
389                 \
390     template <IrisErrorCode (*FUNC)(const ResourceInfo&, const ResourceWriteValue&)>
                \
391     TYPE& setWriteDelegate()
                \
392     {
                \
393         return setWriteDelegate(ResourceWriteDelegate::make<FUNC>());
                \
394     }
                \
395                                                                 \
396       \
397                                       \
398                                                 \
399                 \
400     TYPE& setParentRscId(ResourceId parentRscId)
                \
401     {
                \
402         info->resourceInfo.parentRscId = parentRscId;
                \
403         return *this;
                \
404     }
                \
405                                           \
406                                         \
407     ResourceId getRscId() const
                \
408     {
                \
409         return info->resourceInfo.rscId;
                \
410     }
                \
411                                               \
412                           \
413                                                                 \
414                 \
415     TYPE& getRscId(ResourceId &rscIdOut)
                \
416     {
                \
417         rscIdOut = info->resourceInfo.rscId;
                \
418         return *this;
                \
419     }
420
421 #define INTERNAL_REGISTER_BUILDER_MIXIN(TYPE)
    \
422                                                             \
423                       \
424 \
425     TYPE& setLsbOffset(uint64_t lsbOffset)
      \
426     {
      \
427         info->resourceInfo.registerInfo.lsbOffset = lsbOffset;
      \
428         return *this;
      \
429     }
      \
430                                                           \
431                         \
432                       \
433 \
434     TYPE& setCanonicalRn(uint64_t canonicalRn_)
      \
435     {
      \
436         info->resourceInfo.registerInfo.canonicalRn    = canonicalRn_;
      \
437         info->resourceInfo.registerInfo.hasCanonicalRn = true;
      \
438         return *this;
      \
439     }
      \
440                                     \
441                                     \
442                                 \
443 \
444     TYPE& setCanonicalRnElfDwarf(uint16_t architecture, uint16_t dwarfRegNum)
```

```
    \
445     {
    \
446         if (!instance_builder->canonicalRnSchemeIsAlreadySet) /* Only set property if not already set.
    */\
447         {
    \
448             if (getWithDefault(instance_builder->iris_instance->getPropertyMap(),
    "register.canonicalRnScheme", "").getAsString().empty()) \
449             {
    \
450                 instance_builder->setPropertyCanonicalRnScheme("ElfDwarf");
    \
451             }
    \
452             instance_builder->canonicalRnSchemeIsAlreadySet = true;
    \
453         }
    \
454         return setCanonicalRn(makeCanonicalRnElfDwarf(architecture, dwarfRegNum));
    \
455     }
    \
456                                         \
457                     \
458             \
459                                         \
460 \
461     TYPE& setWriteMask(uint64_t value)
    \
462     {
    \
463         info->resourceInfo.setVector(info->resourceInfo.registerInfo.writeMask, value);
    \
464         return *this;
    \
465     }
    \
466                                           \
467         \
468       \
469                           \
470                     \
471             \
472                         \
473 \
474     template<typename Container>
    \
475     TYPE& setWriteMaskFromContainer(const Container& container)
    \
476     {
    \
477         info->resourceInfo.setVectorFromContainer(info->resourceInfo.registerInfo.writeMask, container);
    \
478         return *this;
    \
479     }
    \
480                                           \
481                   \                                                   \
482                                                                       \
483                                             \
484                                                 \
485 \
486     template<typename T>
    \
487     TYPE& setWriteMask(std::initializer_list<T>&& t)
    \
488     {
    \
489         setWriteMaskFromContainer(std::forward<std::initializer_list<T»(t));
    \
490         return *this;
    \
491     }
    \
492                                         \
493                     \
494             \
495                                         \
496 \
497     TYPE& setResetData(uint64_t value)
    \
498     {
    \
499         info->resourceInfo.setVector(info->resourceInfo.registerInfo.resetData, value);
    \
```

```
500         return *this;
    \
501     }
    \
502                                         \
503         \
504       \
505                           \
506                     \
507             \
508                       \
509 \
510     template<typename Container>
    \
511     TYPE& setResetDataFromContainer(const Container& container)
    \
512     {
    \
513         info->resourceInfo.setVectorFromContainer(info->resourceInfo.registerInfo.resetData, container);
    \
514         return *this;
    \
515     }
    \
516                                         \
517                   \
518                                                                 \
519                                           \
520                                               \
521 \
522     template<typename T>
    \
523     TYPE& setResetData(std::initializer_list<T>&& t)
    \
524     {
    \
525         setResetDataFromContainer(std::forward<std::initializer_list<T»(t));
    \
526         return *this;
    \
527     }
    \
528                                                           \
529                                                     \
530                       \
531 \
532     TYPE& setResetString(const std::string& resetString)
    \
533     {
    \
534         info->resourceInfo.registerInfo.resetString = resetString;
    \
535         return *this;
    \
536     }
    \
537                                                         \
538                     \
539 \
540     TYPE& setAddressOffset(uint64_t addressOffset)
    \
541     {
    \
542         info->resourceInfo.registerInfo.addressOffset    = addressOffset;
    \
543         info->resourceInfo.registerInfo.hasAddressOffset = true;
    \
544         return *this;
    \
545     }
546
547 #define INTERNAL_PARAMETER_BUILDER_MIXIN(TYPE)
    \
548                   \
549                   \
550           \
551                     \
552 \
553     TYPE& setDefaultData(uint64_t value)
    \
554     {
    \
555         info->resourceInfo.setVector(info->resourceInfo.parameterInfo.defaultData, value);
    \
556         return *this;
    \
557     }
```

```
    \
                                  \
       \
      \
                         \
                  \
         \
                    \
\
    template<typename Container>
    \
    TYPE& setDefaultDataFromContainer(const Container& container)
    \
    {
    \
        info->resourceInfo.setVectorFromContainer(info->resourceInfo.parameterInfo.defaultData,
    container); \
        return *this;
    \
    }
    \
                                  \
                \
                                                          \
                                          \
                                              \
\
    template<typename T>
    \
    TYPE& setDefaultData(std::initializer_list<T>&& t)
    \
    {
    \
        setDefaultDataFromContainer(std::forward<std::initializer_list<T>>(t));
    \
        return *this;
    \
    }
    \
          \
                                               \
                   \
\
    TYPE& setDefaultString(const std::string& defaultString)
    \
    {
    \
        info->resourceInfo.parameterInfo.defaultString = defaultString;
    \
        return *this;
    \
    }
    \
                                             \
                                        \
                                    \
\
    TYPE& setInitOnly(bool initOnly = true)
    \
    {
    \
        info->resourceInfo.parameterInfo.initOnly = initOnly;
    \
        /* Implicitly set read-only to make clear that parameter cannot be modified at run-time. */
    \
        info->resourceInfo.rwMode = initOnly ? "r" : std::string(); /* =rw */
    \
        return *this;
    \
    }
    \
                                                          \
             \
    /*                but can still be accessed by resource_getResourceInfo() for clients that know the
    */\
    /*                resource name. */
    \
         \
    TYPE& setHidden(bool hidden = true)
    \
    {
    \
        info->resourceInfo.isHidden = hidden;
    \
        return *this;
    \
    }
```

```
    \
614                                             \
615                     \
616           \
617                                             \
618 \
619     TYPE& setMax(uint64_t value)
    \
620     {
    \
621         info->resourceInfo.setVector(info->resourceInfo.parameterInfo.max, value);
    \
622         return *this;
    \
623     }
    \
624                                         \
625         \
626       \
627                           \
628                     \
629           \
630                       \
631 \
632     template<typename Container>
    \
633     TYPE& setMaxFromContainer(const Container& container)
    \
634     {
    \
635         info->resourceInfo.setVectorFromContainer(info->resourceInfo.parameterInfo.max, container);
    \
636         return *this;
    \
637     }
    \
638                                         \
639                   \
640                                                                   \
641                                               \
642                                                  \
643 \
644     template<typename T>
    \
645     TYPE& setMax(std::initializer_list<T>&& t)
    \
646     {
    \
647         setMaxFromContainer(std::forward<std::initializer_list<T»(t));
    \
648         return *this;
    \
649     }
    \
650                                              \
651                     \
652           \
653                                             \
654 \
655     TYPE& setMin(uint64_t value)
    \
656     {
    \
657         info->resourceInfo.setVector(info->resourceInfo.parameterInfo.min, value);
    \
658         return *this;
    \
659     }
    \
660                                         \
661         \
662       \
663                           \
664                     \
665           \
666                       \
667 \
668     template<typename Container>
    \
669     TYPE& setMinFromContainer(const Container& container)
    \
670     {
    \
671         info->resourceInfo.setVectorFromContainer(info->resourceInfo.parameterInfo.min, container);
    \
672         return *this;
    \
```

```
673     }
    \
674                                     \
675                 \
676                                                             \
677                                         \
678                                             \
679 \
680     template<typename T>
    \
681     TYPE& setMin(std::initializer_list<T>&& t)
    \
682     {
    \
683         setMinFromContainer(std::forward<std::initializer_list<T>>(t));
    \
684         return *this;
    \
685     }
    \
686
690     class ParameterBuilder
691     {
692     private:
693         IrisInstanceResource::ResourceInfoAndAccess* info;
694
695     public:
696         ParameterBuilder(IrisInstanceResource::ResourceInfoAndAccess& info_)
697             : info(&info_)
698         {
699             info->resourceInfo.isParameter = true;
700         }
701
702         ParameterBuilder()
703             : info(nullptr)
704         {
705         }
706
707         INTERNAL_RESOURCE_BUILDER_MIXIN(ParameterBuilder)
708         INTERNAL_PARAMETER_BUILDER_MIXIN(ParameterBuilder)
709     };
710
711     class FieldBuilder;
712
716     class RegisterBuilder
717     {
718     private:
719         IrisInstanceResource::ResourceInfoAndAccess* info{};
720         IrisInstanceResource*                        inst_resource{};
721         IrisInstanceBuilder*                         instance_builder{};
722
723     public:
724         RegisterBuilder(IrisInstanceResource::ResourceInfoAndAccess& info_, IrisInstanceResource*
    inst_resource_, IrisInstanceBuilder *instance_builder_)
725             : info(&info_)
726             , inst_resource(inst_resource_)
727             , instance_builder(instance_builder_)
728         {
729             info->resourceInfo.isRegister = true;
730         }
731
732         RegisterBuilder()
733         {
734         }
735
736         INTERNAL_RESOURCE_BUILDER_MIXIN(RegisterBuilder)
737         INTERNAL_REGISTER_BUILDER_MIXIN(RegisterBuilder)
738
748         FieldBuilder addField(const std::string& name, uint64_t lsbOffset, uint64_t bitWidth, const
    std::string& description);
749
763         FieldBuilder addLogicalField(const std::string& name, uint64_t bitWidth, const std::string&
    description);
764     };
765
769     class FieldBuilder
770     {
771     protected:
772         IrisInstanceResource::ResourceInfoAndAccess* info{};
773         RegisterBuilder*                             parent_reg{};
774         IrisInstanceBuilder*                         instance_builder{};
775
776     public:
777         FieldBuilder(IrisInstanceResource::ResourceInfoAndAccess& info_, RegisterBuilder* parent_reg_,
    IrisInstanceBuilder *instance_builder_)
778             : info(&info_)
779             , parent_reg(parent_reg_)
```

```
780             , instance_builder(instance_builder_)
781         {
782         }
783 
784         FieldBuilder()
785         {
786         }
787 
788         INTERNAL_RESOURCE_BUILDER_MIXIN(FieldBuilder)
789         INTERNAL_REGISTER_BUILDER_MIXIN(FieldBuilder)
790 
795         RegisterBuilder& parent()
796         {
797             return *parent_reg;
798         }
799 
804         FieldBuilder addField(const std::string& name, uint64_t lsbOffset, uint64_t bitWidth, const
    std::string& description)
805         {
806             return parent().addField(name, lsbOffset, bitWidth, description);
807         }
808 
813         FieldBuilder addLogicalField(const std::string& name, uint64_t bitWidth, const std::string&
    description)
814         {
815             return parent().addLogicalField(name, bitWidth, description);
816         }
817     };
818 
819 #undef INTERNAL_RESOURCE_BUILDER_MIXIN
820 #undef INTERNAL_REGISTER_BUILDER_MIXIN
821 #undef INTERNAL_PARAMETER_BUILDER_MIXIN
822 
853     void setDefaultResourceReadDelegate(ResourceReadDelegate delegate = ResourceReadDelegate())
854     {
855         default_reg_read_delegate = delegate;
856     }
857 
885     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, ResourceReadResult&)>
886     void setDefaultResourceReadDelegate(T* instance)
887     {
888         setDefaultResourceReadDelegate(ResourceReadDelegate::make<T, METHOD>(instance));
889     }
890 
910     template <IrisErrorCode (*FUNC)(const ResourceInfo&, ResourceReadResult&)>
911     void setDefaultResourceReadDelegate()
912     {
913         setDefaultResourceReadDelegate(ResourceReadDelegate::make<FUNC>());
914     }
915 
945     void setDefaultResourceWriteDelegate(ResourceWriteDelegate delegate = ResourceWriteDelegate())
946     {
947         default_reg_write_delegate = delegate;
948     }
949 
976     template <typename T, IrisErrorCode (T::*METHOD)(const ResourceInfo&, const ResourceWriteValue&)>
977     void setDefaultResourceWriteDelegate(T* instance)
978     {
979         setDefaultResourceWriteDelegate(ResourceWriteDelegate::make<T, METHOD>(instance));
980     }
981 
1000     template <IrisErrorCode (*FUNC)(const ResourceInfo&, const ResourceWriteValue&)>
1001     void setDefaultResourceWriteDelegate()
1002     {
1003         setDefaultResourceWriteDelegate(ResourceWriteDelegate::make<*FUNC>());
1004     }
1005 
1015     template <typename T, IrisErrorCode (T::*READER)(const ResourceInfo&, ResourceReadResult&),
1016               IrisErrorCode (T::*WRITER)(const ResourceInfo&, const ResourceWriteValue&)>
1017     void setDefaultResourceDelegates(T* instance)
1018     {
1019         setDefaultResourceReadDelegate(ResourceReadDelegate::make<T, READER>(instance));
1020         setDefaultResourceWriteDelegate(ResourceWriteDelegate::make<T, WRITER>(instance));
1021     }
1022 
1045     void beginResourceGroup(const std::string& name,
1046                             const std::string& description,
1047                             uint64_t           subRscIdStart = IRIS_UINT64_MAX,
1048                             const std::string& cname         = std::string());
1049 
1072     ParameterBuilder addParameter(const std::string& name, uint64_t bitWidth, const std::string&
    description);
1073 
1092     ParameterBuilder addStringParameter(const std::string& name, const std::string& description);
1093 
1127     RegisterBuilder addRegister(const std::string& name, uint64_t bitWidth, const std::string&
    description,
```

```
1128                                uint64_t addressOffset = IRIS_UINT64_MAX, uint64_t canonicalRn =
    IRIS_UINT64_MAX);
1129
1148    RegisterBuilder addStringRegister(const std::string& name, const std::string& description);
1149
1170    RegisterBuilder addNoValueRegister(const std::string& name, const std::string& description, const
    std::string& format);
1171
1190    ParameterBuilder enhanceParameter(ResourceId rscId)
1191    {
1192        return ParameterBuilder(*(inst_resource->getResourceInfo(rscId)));
1193    }
1194
1216    RegisterBuilder enhanceRegister(ResourceId rscId)
1217    {
1218        return RegisterBuilder(*(inst_resource->getResourceInfo(rscId)), inst_resource, this);
1219    }
1220
1243    void setPropertyCanonicalRnScheme(const std::string& canonicalRnScheme);
1244
1252    void setNextSubRscId(uint64_t nextSubRscId)
1253    {
1254        inst_resource->setNextSubRscId(nextSubRscId);
1255    }
1256
1266    void setTag(ResourceId rscId, const std::string& tag);
1267
1275    const ResourceInfo &getResourceInfo(ResourceId rscId)
1276    {
1277        return inst_resource->getResourceInfo(rscId)->resourceInfo;
1278    }
1279
1280
1294    class EventSourceBuilder
1295    {
1296    private:
1297        IrisInstanceEvent::EventSourceInfoAndDelegate& info;
1298
1299    public:
1300        EventSourceBuilder(IrisInstanceEvent::EventSourceInfoAndDelegate& info_)
1301            : info(info_)
1302        {
1303        }
1304
1310        EventSourceBuilder& setName(const std::string& name)
1311        {
1312            info.info.name = name;
1313            return *this;
1314        }
1315
1321        EventSourceBuilder& setDescription(const std::string& description)
1322        {
1323            info.info.description = description;
1324            return *this;
1325        }
1326
1332        EventSourceBuilder& setFormat(const std::string& format)
1333        {
1334            info.info.format = format;
1335            return *this;
1336        }
1337
1343        EventSourceBuilder& setCounter(bool counter = true)
1344        {
1345            info.info.counter = counter;
1346            return *this;
1347        }
1348
1356        EventSourceBuilder& setHidden(bool hidden = true)
1357        {
1358            info.info.isHidden = hidden;
1359            return *this;
1360        }
1361
1368        EventSourceBuilder& hasSideEffects(bool hasSideEffects_ = true)
1369        {
1370            info.info.hasSideEffects = hasSideEffects_;
1371            return *this;
1372        }
1373
1386        EventSourceBuilder& addField(const std::string& name, const std::string& type, uint64_t
    sizeInBytes,
1387                                     const std::string& description)
1388        {
1389            info.info.addField(name, type, sizeInBytes, description);
1390            return *this;
1391        }
```

```
1392
1403        EventSourceBuilder& addEnumElement(uint64_t value, const std::string& symbol, const
    std::string& description = "")
1404        {
1405            if (info.info.fields.size() > 0)
1406            {
1407                info.info.fields.back().addEnumElement(value, symbol, description);
1408                return *this;
1409            }
1410            else
1411            {
1412                throw IrisInternalError("EventSourceInfo has no fields to add an enum element to.");
1413            }
1414        }
1415
1425        EventSourceBuilder& setEventStreamCreateDelegate(EventStreamCreateDelegate delegate)
1426        {
1427            info.createEventStream = delegate;
1428            return *this;
1429        }
1430
1443        template <typename T,
1444                  IrisErrorCode (T::*METHOD)(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&)>
1445        EventSourceBuilder& setEventStreamCreateDelegate(T* instance)
1446        {
1447            return setEventStreamCreateDelegate(EventStreamCreateDelegate::make<T, METHOD>(instance));
1448        }
1449
1463        template<typename T>
1464        EventSourceBuilder& addOption(const std::string& name, const std::string& type, const T&
    defaultValue,
1465                                      bool optional, const std::string& description)
1466        {
1467            info.info.addOption(name, type, defaultValue, optional, description);
1468            return *this;
1469        }
1470    };
1471
1486    EventSourceBuilder addEventSource(const std::string& name, bool isHidden = false)
1487    {
1488        return EventSourceBuilder(inst_event->addEventSource(name, isHidden));
1489    }
1490
1502    EventSourceBuilder addEventSource(const std::string& name, IrisEventEmitterBase& event_emitter,
    bool isHidden = false)
1503    {
1504        IrisInstanceEvent::EventSourceInfoAndDelegate& info = inst_event->addEventSource(name,
    isHidden);
1505
1506        event_emitter.setIrisInstance(iris_instance);
1507        event_emitter.setEvSrcId(info.info.evSrcId);
1508        info.createEventStream = EventStreamCreateDelegate::make<IrisEventEmitterBase,
1509
    &IrisEventEmitterBase::createEventStream>(&event_emitter);
1510
1511        return EventSourceBuilder(info);
1512    }
1513
1523    EventSourceBuilder enhanceEventSource(const std::string& name)
1524    {
1525        IrisInstanceEvent::EventSourceInfoAndDelegate& info = inst_event->enhanceEventSource(name);
1526        return EventSourceBuilder(info);
1527    }
1528
1534    void deleteEventSource(const std::string& name)
1535    {
1536        inst_event->deleteEventSource(name);
1537    }
1538
1545    bool hasEventSource(const std::string& name)
1546    {
1547        return inst_event->hasEventSource(name);
1548    }
1549
1575    EventSourceBuilder setRegisterReadEvent(const std::string& name, const std::string& description =
    std::string());
1576
1602    EventSourceBuilder setRegisterReadEvent(const std::string& name, IrisRegisterEventEmitterBase&
    event_emitter);
1603
1610    void finalizeRegisterReadEvent();
1611
1638    EventSourceBuilder setRegisterUpdateEvent(const std::string& name, const std::string& description =
    std::string());
1639
1666    EventSourceBuilder setRegisterUpdateEvent(const std::string& name, IrisRegisterEventEmitterBase&
```

```
    event_emitter);
1667
1674    void finalizeRegisterUpdateEvent();
1675
1682    void resetRegisterReadEvent();
1683
1690    void resetRegisterUpdateEvent();
1691
1723    void setDefaultEsCreateDelegate(EventStreamCreateDelegate delegate)
1724    {
1725        inst_event->setDefaultEsCreateDelegate(delegate);
1726    }
1727
1758    template <typename T, IrisErrorCode (T::*METHOD)(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&)>
1759    void setDefaultEsCreateDelegate(T* instance)
1760    {
1761        setDefaultEsCreateDelegate(EventStreamCreateDelegate::make<T, METHOD>(instance));
1762    }
1763
1786    template <IrisErrorCode (*FUNC)(EventStream*&, const EventSourceInfo&, const
    std::vector<std::string>&)>
1787    void setDefaultEsCreateDelegate()
1788    {
1789        setDefaultEsCreateDelegate(EventStreamCreateDelegate::make<FUNC>());
1790    }
1791
1798    IrisInstanceEvent* getIrisInstanceEvent() { return inst_event; }
1799
1831    void setBreakpointSetDelegate(BreakpointSetDelegate delegate)
1832    {
1833        if (inst_breakpoint.getPtr() == nullptr)
1834        {
1835            // Ensure the underlying IrisInstanceBreakpoint object is initialised too.
1836            inst_breakpoint.init();
1837        }
1838        user_setBreakpoint = delegate;
1839    }
1840
1862    template <typename T, IrisErrorCode (T::*METHOD)(BreakpointInfo&)>
1863    void setBreakpointSetDelegate(T* instance)
1864    {
1865        setBreakpointSetDelegate(BreakpointSetDelegate::make<T, METHOD>(instance));
1866    }
1867
1881    template <IrisErrorCode (*FUNC)(BreakpointInfo&)>
1882    void setBreakpointSetDelegate()
1883    {
1884        setBreakpointSetDelegate(BreakpointSetDelegate::make<FUNC>());
1885    }
1886
1908    void setBreakpointDeleteDelegate(BreakpointDeleteDelegate delegate)
1909    {
1910        if (inst_breakpoint.getPtr() == nullptr)
1911        {
1912            // Ensure the underlying IrisInstanceBreakpoint object is initialised too.
1913            inst_breakpoint.init();
1914        }
1915        user_deleteBreakpoint = delegate;
1916    }
1917
1939    template <typename T, IrisErrorCode (T::*METHOD)(const BreakpointInfo&)>
1940    void setBreakpointDeleteDelegate(T* instance)
1941    {
1942        setBreakpointDeleteDelegate(BreakpointDeleteDelegate::make<T, METHOD>(instance));
1943    }
1944
1958    template <IrisErrorCode (*FUNC)(const BreakpointInfo&)>
1959    void setBreakpointDeleteDelegate()
1960    {
1961        setBreakpointDeleteDelegate(BreakpointDeleteDelegate::make<FUNC>());
1962    }
1963
1985    void setHandleBreakpointHitDelegate(HandleBreakpointHitDelegate delegate)
1986    {
1987        if (inst_breakpoint.getPtr() == nullptr)
1988        {
1989            // Ensure the underlying IrisInstanceBreakpoint object is initialised too.
1990            inst_breakpoint.init();
1991        }
1992
1993        inst_breakpoint->setHandleBreakpointHitDelegate(delegate);
1994    }
1995
2017    template <typename T, IrisErrorCode (T::*METHOD)(const BreakpointHitInfo&)>
2018    void setHandleBreakpointHitDelegate(T* instance)
2019    {
```

```
2020         setHandleBreakpointHitDelegate(HandleBreakpointHitDelegate::make<T, METHOD>(instance));
2021     }
2022
2036     template <IrisErrorCode (*FUNC)(const BreakpointHitInfo&)>
2037     void setHandleBreakpointHitDelegate()
2038     {
2039         setHandleBreakpointHitDelegate(HandleBreakpointHitDelegate::make<FUNC>());
2040     }
2041
2052     void notifyBreakpointHit(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId pcSpaceId)
2053     {
2054         inst_breakpoint->notifyBreakpointHit(bptId, time, pc, pcSpaceId);
2055     }
2056
2072     void notifyBreakpointHitData(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
2073                                  uint64_t accessAddr, uint64_t accessSize,
2074                                  const std::string& accessRw, const std::vector<uint64_t>& data)
2075     {
2076         inst_breakpoint->notifyBreakpointHitData(bptId, time, pc, pcSpaceId, accessAddr, accessSize,
    accessRw, data);
2077     }
2078
2092     void notifyBreakpointHitRegister(BreakpointId bptId, uint64_t time, uint64_t pc, MemorySpaceId
    pcSpaceId,
2093                                      const std::string& accessRw, const std::vector<uint64_t>& data)
2094     {
2095         inst_breakpoint->notifyBreakpointHitRegister(bptId, time, pc, pcSpaceId, accessRw, data);
2096     }
2097
2105     const BreakpointInfo* getBreakpointInfo(BreakpointId bptId)
2106     {
2107         return inst_breakpoint->getBreakpointInfo(bptId);
2108     }
2109
2111     void addBreakpointCondition(const std::string& name, const std::string& type, const std::string&
    description,
2112                                 const std::vector<std::string> bpt_types = std::vector<std::string>())
2113     {
2114         inst_breakpoint->addCondition(name, type, description, bpt_types);
2115     }
2116
2130     class MemorySpaceBuilder
2131     {
2132     private:
2133         IrisInstanceMemory::SpaceInfoAndAccess& info;
2134
2135     public:
2136         MemorySpaceBuilder(IrisInstanceMemory::SpaceInfoAndAccess& info_)
2137             : info(info_)
2138         {
2139         }
2140
2147         MemorySpaceBuilder& setName(const std::string& name)
2148         {
2149             info.spaceInfo.name = name;
2150             return *this;
2151         }
2152
2159         MemorySpaceBuilder& setDescription(const std::string& description)
2160         {
2161             info.spaceInfo.description = description;
2162             return *this;
2163         }
2164
2171         MemorySpaceBuilder& setMinAddr(uint64_t minAddr)
2172         {
2173             info.spaceInfo.minAddr = minAddr;
2174             return *this;
2175         }
2176
2183         MemorySpaceBuilder& setMaxAddr(uint64_t maxAddr)
2184         {
2185             info.spaceInfo.maxAddr = maxAddr;
2186             return *this;
2187         }
2188
2195         MemorySpaceBuilder& setCanonicalMsn(uint64_t canonicalMsn)
2196         {
2197             info.spaceInfo.canonicalMsn = canonicalMsn;
2198             return *this;
2199         }
2200
2207         MemorySpaceBuilder& setEndianness(const std::string& endianness)
2208         {
2209             info.spaceInfo.endianness = endianness;
2210             return *this;
```

```
2211        }
2212
2220        MemorySpaceBuilder& addAttribute(const std::string& name, AttributeInfo attrib)
2221        {
2222            info.spaceInfo.attrib[name] = attrib;
2223            return *this;
2224        }
2225
2233        MemorySpaceBuilder& setAttributeDefault(const std::string& name, IrisValue value)
2234        {
2235            info.spaceInfo.attribDefaults[name] = value;
2236            return *this;
2237        }
2238
2251        MemorySpaceBuilder& setSupportedByteWidths(uint64_t supportedByteWidths)
2252        {
2253            info.spaceInfo.supportedByteWidths = supportedByteWidths;
2254            return *this;
2255        }
2256
2267        MemorySpaceBuilder& setReadDelegate(MemoryReadDelegate delegate)
2268        {
2269            info.readDelegate = delegate;
2270            return *this;
2271        }
2272
2283        MemorySpaceBuilder& setWriteDelegate(MemoryWriteDelegate delegate)
2284        {
2285            info.writeDelegate = delegate;
2286            return *this;
2287        }
2288
2299        MemorySpaceBuilder& setSidebandDelegate(MemoryGetSidebandInfoDelegate delegate)
2300        {
2301            info.sidebandDelegate = delegate;
2302            return *this;
2303        }
2304
2318        template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
    uint64_t, const AttributeValueMap&, MemoryReadResult&)>
2319        MemorySpaceBuilder& setReadDelegate(T* instance)
2320        {
2321            return setReadDelegate(MemoryReadDelegate::make<T, METHOD>(instance));
2322        }
2323
2337        template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
    uint64_t, const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2338        MemorySpaceBuilder& setWriteDelegate(T* instance)
2339        {
2340            return setWriteDelegate(MemoryWriteDelegate::make<T, METHOD>(instance));
2341        }
2342
2356        template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, const
    IrisValueMap&, const std::vector<std::string>&, IrisValueMap&)>
2357        MemorySpaceBuilder& setSidebandDelegate(T* instance)
2358        {
2359            return setSidebandDelegate(MemoryGetSidebandInfoDelegate::make<T, METHOD>(instance));
2360        }
2361
2372        template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2373                                        const AttributeValueMap&, MemoryReadResult&)>
2374        MemorySpaceBuilder& setReadDelegate()
2375        {
2376            return setReadDelegate(MemoryReadDelegate::make<FUNC>());
2377        }
2378
2389        template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2390                                        const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2391        MemorySpaceBuilder& setWriteDelegate()
2392        {
2393            return setWriteDelegate(MemoryWriteDelegate::make<FUNC>());
2394        }
2395
2406        template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, const IrisValueMap&,
2407                                        const std::vector<std::string>&, IrisValueMap&)>
2408        MemorySpaceBuilder& setSidebandDelegate()
2409        {
2410            return setSidebandDelegate(MemoryGetSidebandInfoDelegate::make<FUNC>());
2411        }
2412
2421        MemorySpaceId getSpaceId() const
2422        {
2423            return info.spaceInfo.spaceId;
2424        }
2425    };
2426
2430    class AddressTranslationBuilder
```

```
2431     {
2432     private:
2433         IrisInstanceMemory::AddressTranslationInfoAndAccess& info;
2434 
2435     public:
2436         AddressTranslationBuilder(IrisInstanceMemory::AddressTranslationInfoAndAccess& info_)
2437             : info(info_)
2438         {
2439         }
2440 
2451         AddressTranslationBuilder& setTranslateDelegate(MemoryAddressTranslateDelegate delegate)
2452         {
2453             info.translateDelegate = delegate;
2454             return *this;
2455         }
2456 
2470         template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, uint64_t, uint64_t,
     MemoryAddressTranslationResult&)>
2471         AddressTranslationBuilder& setTranslateDelegate(T* instance)
2472         {
2473             return setTranslateDelegate(MemoryAddressTranslateDelegate::make<T, METHOD>(instance));
2474         }
2475 
2486         template <IrisErrorCode (*FUNC)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult&)>
2487         AddressTranslationBuilder& setTranslateDelegate()
2488         {
2489             return setTranslateDelegate(MemoryAddressTranslateDelegate::make<FUNC>());
2490         }
2491     };
2492 
2505     void setPropertyCanonicalMsnScheme(const std::string& canonicalMsnScheme);
2506 
2539     void setDefaultMemoryReadDelegate(MemoryReadDelegate delegate = MemoryReadDelegate())
2540     {
2541         inst_memory->setDefaultReadDelegate(delegate);
2542     }
2543 
2576     template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
     uint64_t, const AttributeValueMap&, MemoryReadResult&)>
2577     void setDefaultMemoryReadDelegate(T* instance)
2578     {
2579         setDefaultMemoryReadDelegate(MemoryReadDelegate::make<T, METHOD>(instance));
2580     }
2581 
2607     template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2608                                     const AttributeValueMap&, MemoryReadResult&)>
2609     void setDefaultMemoryReadDelegate()
2610     {
2611         setDefaultMemoryReadDelegate(MemoryReadDelegate::make<FUNC>());
2612     }
2613 
2647     void setDefaultMemoryWriteDelegate(MemoryWriteDelegate delegate = MemoryWriteDelegate())
2648     {
2649         inst_memory->setDefaultWriteDelegate(delegate);
2650     }
2651 
2685     template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, uint64_t,
     uint64_t, const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2686     void setDefaultMemoryWriteDelegate(T* instance)
2687     {
2688         setDefaultMemoryWriteDelegate(MemoryWriteDelegate::make<T, METHOD>(instance));
2689     }
2690 
2716     template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, uint64_t, uint64_t,
2717                                     const AttributeValueMap&, const uint64_t*, MemoryWriteResult&)>
2718     void setDefaultMemoryWriteDelegate()
2719     {
2720         setDefaultMemoryWriteDelegate(MemoryWriteDelegate::make<FUNC>());
2721     }
2722 
2741     MemorySpaceBuilder addMemorySpace(const std::string& name)
2742     {
2743         return MemorySpaceBuilder(inst_memory->addMemorySpace(name));
2744     }
2745 
2777     void setDefaultAddressTranslateDelegate(MemoryAddressTranslateDelegate delegate =
     MemoryAddressTranslateDelegate())
2778     {
2779         inst_memory->setDefaultTranslateDelegate(delegate);
2780     }
2781 
2809     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, uint64_t, uint64_t,
     MemoryAddressTranslationResult&)>
2810     void setDefaultAddressTranslateDelegate(T* instance)
2811     {
2812         setDefaultAddressTranslateDelegate(MemoryAddressTranslateDelegate::make<T, METHOD>(instance));
2813     }
```

```
2814
2834     template <IrisErrorCode (*FUNC)(uint64_t, uint64_t, uint64_t, MemoryAddressTranslationResult&)>
2835     void setDefaultAddressTranslateDelegate()
2836     {
2837         setDefaultAddressTranslateDelegate(MemoryAddressTranslateDelegate::make<FUNC>());
2838     }
2839
2856     AddressTranslationBuilder addAddressTranslation(MemorySpaceId inSpaceId, MemorySpaceId outSpaceId,
2857                                                     const std::string& description)
2858     {
2859         return AddressTranslationBuilder(inst_memory->addAddressTranslation(inSpaceId, outSpaceId,
     description));
2860     }
2861
2894     void setDefaultGetMemorySidebandInfoDelegate(MemoryGetSidebandInfoDelegate delegate)
2895     {
2896         inst_memory->setDefaultGetSidebandInfoDelegate(delegate);
2897     }
2898
2927     template <typename T, IrisErrorCode (T::*METHOD)(const MemorySpaceInfo&, uint64_t, const
     IrisValueMap&, const std::vector<std::string>&, IrisValueMap&)>
2928     void setDefaultGetMemorySidebandInfoDelegate(T* instance)
2929     {
2930         setDefaultGetMemorySidebandInfoDelegate(MemoryGetSidebandInfoDelegate::make<T,
     METHOD>(instance));
2931     }
2932
2953     template <IrisErrorCode (*FUNC)(const MemorySpaceInfo&, uint64_t, const IrisValueMap&,
2954                                     const std::vector<std::string>&, IrisValueMap&)>
2955     void setDefaultGetMemorySidebandInfoDelegate()
2956     {
2957         setDefaultGetMemorySidebandInfoDelegate(MemoryGetSidebandInfoDelegate::make<FUNC>());
2958     }
2959
2994     void setLoadImageFileDelegate(ImageLoadFileDelegate delegate = ImageLoadFileDelegate())
2995     {
2996         inst_image->setLoadImageFileDelegate(delegate);
2997     }
2998
3019     template <typename T, IrisErrorCode (T::*METHOD)(const std::string&)>
3020     void setLoadImageFileDelegate(T* instance)
3021     {
3022         setLoadImageFileDelegate(ImageLoadFileDelegate::make<T, METHOD>(instance));
3023     }
3024
3037     template <IrisErrorCode (*FUNC)(const std::string&)>
3038     void setLoadImageFileDelegate()
3039     {
3040         setLoadImageFileDelegate(ImageLoadFileDelegate::make<FUNC>());
3041     }
3042
3067     void setLoadImageDataDelegate(ImageLoadDataDelegate delegate = ImageLoadDataDelegate())
3068     {
3069         inst_image->setLoadImageDataDelegate(delegate);
3070     }
3071
3092     template <typename T, IrisErrorCode (T::*METHOD)(const std::vector<uint8_t>&)>
3093     void setLoadImageDataDelegate(T* instance)
3094     {
3095         setLoadImageDataDelegate(ImageLoadDataDelegate::make<T, METHOD>(instance));
3096     }
3097
3110     template <IrisErrorCode (*FUNC)(const std::vector<uint8_t>&)>
3111     void setLoadImageDataDelegate()
3112     {
3113         setLoadImageDataDelegate(ImageLoadDataDelegate::make<FUNC>());
3114     }
3115
3131     uint64_t openImage(const std::string& filename)
3132     {
3133         return inst_image_cb->openImage(filename);
3134     }
3135
3170     void setRemainingStepSetDelegate(RemainingStepSetDelegate delegate = RemainingStepSetDelegate())
3171     {
3172         inst_step->setRemainingStepSetDelegate(delegate);
3173     }
3174
3199     void setRemainingStepGetDelegate(RemainingStepGetDelegate delegate)
3200     {
3201         inst_step->setRemainingStepGetDelegate(delegate);
3202     }
3203
3224     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, const std::string&)>
3225     void setRemainingStepSetDelegate(T* instance)
3226     {
3227         setRemainingStepSetDelegate(RemainingStepSetDelegate::make<T, METHOD>(instance));
```

```
3228     }
3229
3250     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t&, const std::string&)>
3251     void setRemainingStepGetDelegate(T* instance)
3252     {
3253         setRemainingStepGetDelegate(RemainingStepGetDelegate::make<T, METHOD>(instance));
3254     }
3255
3268     template <IrisErrorCode (*FUNC)(uint64_t, const std::string&)>
3269     void setRemainingStepSetDelegate()
3270     {
3271         setRemainingStepSetDelegate(RemainingStepSetDelegate::make<FUNC>());
3272     }
3273
3286     template <IrisErrorCode (*FUNC)(uint64_t&, const std::string&)>
3287     void setRemainingStepGetDelegate()
3288     {
3289         setRemainingStepGetDelegate(RemainingStepGetDelegate::make<FUNC>());
3290     }
3291
3316     //
3317     void setStepCountGetDelegate(StepCountGetDelegate delegate = StepCountGetDelegate())
3318     {
3319         inst_step->setStepCountGetDelegate(delegate);
3320     }
3321
3342     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t&, const std::string&)>
3343     void setStepCountGetDelegate(T* instance)
3344     {
3345         setStepCountGetDelegate(RemainingStepGetDelegate::make<T, METHOD>(instance));
3346     }
3347
3360     template <IrisErrorCode (*FUNC)(uint64_t&, const std::string&)>
3361     void setStepCountGetDelegate()
3362     {
3363         setStepCountGetDelegate(RemainingStepGetDelegate::make<FUNC>());
3364     }
3365
3370     /*
3371      * @brief exec_apis IrisInstanceBuilder per-instance execution APIs
3372      * @{
3373      */
3374
3399     void setExecutionStateSetDelegate(PerInstanceExecutionStateSetDelegate delegate =
    PerInstanceExecutionStateSetDelegate())
3400     {
3401         inst_per_inst_exec->setExecutionStateSetDelegate(delegate);
3402     }
3403
3424     template <typename T, IrisErrorCode (T::*METHOD)(bool)>
3425     void setExecutionStateSetDelegate(T* instance)
3426     {
3427         setExecutionStateSetDelegate(PerInstanceExecutionStateSetDelegate::make<T, METHOD>(instance));
3428     }
3429
3442     template <IrisErrorCode (*FUNC)(bool)>
3443     void setExecutionStateSetDelegate()
3444     {
3445         setExecutionStateSetDelegate(PerInstanceExecutionStateSetDelegate::make<FUNC>());
3446     }
3447
3472     void setExecutionStateGetDelegate(PerInstanceExecutionStateGetDelegate delegate)
3473     {
3474         inst_per_inst_exec->setExecutionStateGetDelegate(delegate);
3475     }
3476
3497     template <typename T, IrisErrorCode (T::*METHOD)(bool&)>
3498     void setExecutionStateGetDelegate(T* instance)
3499     {
3500         setExecutionStateGetDelegate(PerInstanceExecutionStateGetDelegate::make<T, METHOD>(instance));
3501     }
3502
3515     template <IrisErrorCode (*FUNC)(bool&)>
3516     void setExecutionStateGetDelegate()
3517     {
3518         setExecutionStateGetDelegate(PerInstanceExecutionStateGetDelegate::make<FUNC>());
3519     }
3520
3525     /*
3526      * @brief table_apis IrisInstanceBuilder table APIs
3527      * @{
3528      */
3529
3530     class TableColumnBuilder;
3531
3535     class TableBuilder
3536     {
```

```
3537    private:
3538        IrisInstanceTable::TableInfoAndAccess& info;
3539
3540    public:
3541        TableBuilder(IrisInstanceTable::TableInfoAndAccess& info_)
3542            : info(info_)
3543        {
3544        }
3545
3551        TableBuilder& setName(const std::string& name)
3552        {
3553            info.tableInfo.name = name;
3554            return *this;
3555        }
3556
3562        TableBuilder& setDescription(const std::string& description)
3563        {
3564            info.tableInfo.description = description;
3565            return *this;
3566        }
3567
3573        TableBuilder& setMinIndex(uint64_t minIndex)
3574        {
3575            info.tableInfo.minIndex = minIndex;
3576            return *this;
3577        }
3578
3584        TableBuilder& setMaxIndex(uint64_t maxIndex)
3585        {
3586            info.tableInfo.maxIndex = maxIndex;
3587            return *this;
3588        }
3589
3595        TableBuilder& setIndexFormatHint(const std::string& hint)
3596        {
3597            info.tableInfo.indexFormatHint = hint;
3598            return *this;
3599        }
3600
3606        TableBuilder& setFormatShort(const std::string& format)
3607        {
3608            info.tableInfo.formatShort = format;
3609            return *this;
3610        }
3611
3617        TableBuilder& setFormatLong(const std::string& format)
3618        {
3619            info.tableInfo.formatLong = format;
3620            return *this;
3621        }
3622
3632        TableBuilder& setReadDelegate(TableReadDelegate delegate)
3633        {
3634            info.readDelegate = delegate;
3635            return *this;
3636        }
3637
3647        TableBuilder& setWriteDelegate(TableWriteDelegate delegate)
3648        {
3649            info.writeDelegate = delegate;
3650            return *this;
3651        }
3652
3664        template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, uint64_t, uint64_t,
    TableReadResult&)>
3665        TableBuilder& setReadDelegate(T* instance)
3666        {
3667            return setReadDelegate(TableReadDelegate::make<T, METHOD>(instance));
3668        }
3669
3681        template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, const TableRecords&,
    TableWriteResult&)>
3682        TableBuilder& setWriteDelegate(T* instance)
3683        {
3684            return setWriteDelegate(TableWriteDelegate::make<T, METHOD>(instance));
3685        }
3686
3696        template <IrisErrorCode (*FUNC)(const TableInfo&, uint64_t, uint64_t, TableReadResult&)>
3697        TableBuilder& setReadDelegate()
3698        {
3699            return setReadDelegate(TableReadDelegate::make<FUNC>());
3700        }
3701
3711        template <IrisErrorCode (*FUNC)(const TableInfo&, const TableRecords&, TableWriteResult&)>
3712        TableBuilder& setWriteDelegate()
3713        {
3714            return setWriteDelegate(TableWriteDelegate::make<FUNC>());
```

```
3715         }
3716
3727         TableBuilder& addColumnInfo(const TableColumnInfo& columnInfo)
3728         {
3729             info.tableInfo.columns.push_back(columnInfo);
3730             return *this;
3731         }
3732
3744         TableColumnBuilder addColumn(const std::string& name);
3745     };
3746
3750     class TableColumnBuilder
3751     {
3752     private:
3753         TableBuilder&    parent;
3754         TableColumnInfo& info;
3755
3756     public:
3757         TableColumnBuilder(TableBuilder& parent_, TableColumnInfo& info_)
3758             : parent(parent_)
3759             , info(info_)
3760         {
3761         }
3762
3772         TableBuilder& addColumnInfo(const TableColumnInfo& columnInfo)
3773         {
3774             return parent.addColumnInfo(columnInfo);
3775         }
3776
3788         TableColumnBuilder addColumn(const std::string& name) { return parent.addColumn(name); }
3789
3798         TableBuilder& endColumn()
3799         {
3800             return parent;
3801         }
3802
3809         TableColumnBuilder& setName(const std::string& name)
3810         {
3811             info.name = name;
3812             return *this;
3813         }
3814
3821         TableColumnBuilder& setDescription(const std::string& description)
3822         {
3823             info.description = description;
3824             return *this;
3825         }
3826
3833         TableColumnBuilder& setFormat(const std::string& format)
3834         {
3835             info.format = format;
3836             return *this;
3837         }
3838
3845         TableColumnBuilder& setType(const std::string& type)
3846         {
3847             info.type = type;
3848             return *this;
3849         }
3850
3857         TableColumnBuilder& setBitWidth(uint64_t bitWidth)
3858         {
3859             info.bitWidth = bitWidth;
3860             return *this;
3861         }
3862
3869         TableColumnBuilder& setFormatShort(const std::string& format)
3870         {
3871             info.formatShort = format;
3872             return *this;
3873         }
3874
3881         TableColumnBuilder& setFormatLong(const std::string& format)
3882         {
3883             info.formatLong = format;
3884             return *this;
3885         }
3886
3893         TableColumnBuilder& setRwMode(const std::string& rwMode)
3894         {
3895             info.rwMode = rwMode;
3896             return *this;
3897         }
3898     };
3899
3922     TableBuilder addTable(const std::string& name)
3923     {
```

```
3924         return TableBuilder(inst_table->addTableInfo(name));
3925     }
3926
3957     void setDefaultTableReadDelegate(TableReadDelegate delegate = TableReadDelegate())
3958     {
3959         inst_table->setDefaultReadDelegate(delegate);
3960     }
3961
3993     void setDefaultTableWriteDelegate(TableWriteDelegate delegate = TableWriteDelegate())
3994     {
3995         inst_table->setDefaultWriteDelegate(delegate);
3996     }
3997
4024     template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, uint64_t, uint64_t,
    TableReadResult&)>
4025     void setDefaultTableReadDelegate(T* instance)
4026     {
4027         setDefaultTableReadDelegate(TableReadDelegate::make<T, METHOD>(instance));
4028     }
4029
4057     template <typename T, IrisErrorCode (T::*METHOD)(const TableInfo&, const TableRecords&,
    TableWriteResult&)>
4058     void setDefaultTableWriteDelegate(T* instance)
4059     {
4060         setDefaultTableWriteDelegate(TableWriteDelegate::make<T, METHOD>(instance));
4061     }
4062
4081     template <IrisErrorCode (*FUNC)(const TableInfo&, uint64_t, uint64_t, TableReadResult&)>
4082     void setDefaultTableReadDelegate()
4083     {
4084         setDefaultTableReadDelegate(TableReadDelegate::make<FUNC>());
4085     }
4086
4106     template <IrisErrorCode (*FUNC)(const TableInfo&, const TableRecords&, TableWriteResult&)>
4107     void setDefaultTableWriteDelegate()
4108     {
4109         setDefaultTableWriteDelegate(TableWriteDelegate::make<FUNC>());
4110     }
4111
4122     void setGetCurrentDisassemblyModeDelegate(GetCurrentDisassemblyModeDelegate delegate)
4123     {
4124         inst_disass->setGetCurrentModeDelegate(delegate);
4125     }
4126
4127     template <typename T, IrisErrorCode (T::*METHOD)(std::string&)>
4128     void setGetCurrentDisassemblyModeDelegate(T* instance)
4129     {
4130         setGetCurrentDisassemblyModeDelegate(GetCurrentDisassemblyModeDelegate::make<T,
    METHOD>(instance));
4131     }
4132
4134     void setGetDisassemblyDelegate(GetDisassemblyDelegate delegate)
4135     {
4136         inst_disass->setGetDisassemblyDelegate(delegate);
4137     }
4138
4139     template <typename T, IrisErrorCode (T::*METHOD)(uint64_t, const std::string&, MemoryReadResult&,
    uint64_t, uint64_t, std::vector<DisassemblyLine>&)>
4140     void setGetDisassemblyDelegate(T* instance)
4141     {
4142         setGetDisassemblyDelegate(GetDisassemblyDelegate::make<T, METHOD>(instance));
4143     }
4144
4145     template <IrisErrorCode (*FUNC)(uint64_t, const std::string&, MemoryReadResult&,
4146                                     uint64_t, uint64_t, std::vector<DisassemblyLine>&)>
4147     void setGetDisassemblyDelegate()
4148     {
4149         setGetDisassemblyDelegate(GetDisassemblyDelegate::make<FUNC>());
4150     }
4151
4153     void setDisassembleOpcodeDelegate(DisassembleOpcodeDelegate delegate)
4154     {
4155         inst_disass->setDisassembleOpcodeDelegate(delegate);
4156     }
4157
4158     template <typename T, IrisErrorCode (T::*METHOD)(const std::vector<uint64_t>&, uint64_t, const
    std::string&, DisassembleContext&, DisassemblyLine&)>
4159     void setDisassembleOpcodeDelegate(T* instance)
4160     {
4161         setDisassembleOpcodeDelegate(DisassembleOpcodeDelegate::make<T, METHOD>(instance));
4162     }
4163
4164     template <IrisErrorCode (*FUNC)(const std::vector<uint64_t>&, uint64_t, const std::string&,
4165                                     DisassembleContext&, DisassemblyLine&)>
4166     void setDisassembleOpcodeDelegate()
4167     {
4168         setDisassembleOpcodeDelegate(DisassembleOpcodeDelegate::make<FUNC>());
```

```
4169     }
4170
4172     void addDisassemblyMode(const std::string& name, const std::string& description)
4173     {
4174         inst_disass->addDisassemblyMode(name, description);
4175     }
4176
4210     void setDbgStateSetRequestDelegate(DebuggableStateSetRequestDelegate delegate =
     DebuggableStateSetRequestDelegate())
4211     {
4212         inst_dbg_state->setSetRequestDelegate(delegate);
4213     }
4214
4235     template <typename T, IrisErrorCode (T::*METHOD)(bool)>
4236     void setDbgStateSetRequestDelegate(T* instance)
4237     {
4238         setDbgStateSetRequestDelegate(DebuggableStateSetRequestDelegate::make<T, METHOD>(instance));
4239     }
4240
4253     template <IrisErrorCode (*FUNC)(bool)>
4254     void setDbgStateSetRequestDelegate()
4255     {
4256         setDbgStateSetRequestDelegate(DebuggableStateSetRequestDelegate::make<FUNC>());
4257     }
4258
4283     void setDbgStateGetAcknowledgeDelegate(DebuggableStateGetAcknowledgeDelegate delegate =
     DebuggableStateGetAcknowledgeDelegate())
4284     {
4285         inst_dbg_state->setGetAcknowledgeDelegate(delegate);
4286     }
4287
4308     template <typename T, IrisErrorCode (T::*METHOD)(bool&)>
4309     void setDbgStateGetAcknowledgeDelegate(T* instance)
4310     {
4311         setDbgStateGetAcknowledgeDelegate(DebuggableStateGetAcknowledgeDelegate::make<T,
     METHOD>(instance));
4312     }
4313
4326     template <IrisErrorCode (*FUNC)(bool&)>
4327     void setDbgStateGetAcknowledgeDelegate()
4328     {
4329         setDbgStateGetAcknowledgeDelegate(DebuggableStateGetAcknowledgeDelegate::make<FUNC>());
4330     }
4331
4359     template <typename T, IrisErrorCode (T::*SET_REQUEST)(bool), IrisErrorCode
     (T::*GET_ACKNOWLEDGE)(bool&)>
4360     void setDbgStateDelegates(T* instance)
4361     {
4362         setDbgStateSetRequestDelegate<T, SET_REQUEST>(instance);
4363         setDbgStateGetAcknowledgeDelegate<T, GET_ACKNOWLEDGE>(instance);
4364     }
4365
4367     void setCheckpointSaveDelegate(CheckpointSaveDelegate delegate = CheckpointSaveDelegate())
4368     {
4369         inst_checkpoint->setCheckpointSaveDelegate(delegate);
4370     }
4371
4372     template <typename T, IrisErrorCode (T::*METHOD)(const std::string&)>
4373     void setCheckpointSaveDelegate(T* instance)
4374     {
4375         setCheckpointSaveDelegate(CheckpointSaveDelegate::make<T, METHOD>(instance));
4376     }
4377
4378     void setCheckpointRestoreDelegate(CheckpointRestoreDelegate delegate = CheckpointRestoreDelegate())
4379     {
4380         inst_checkpoint->setCheckpointRestoreDelegate(delegate);
4381     }
4382
4383     template <typename T, IrisErrorCode (T::*METHOD)(const std::string&)>
4384     void setCheckpointRestoreDelegate(T* instance)
4385     {
4386         setCheckpointRestoreDelegate(CheckpointRestoreDelegate::make<T, METHOD>(instance));
4387     }
4388
4401     class SemihostingManager
4402     {
4403     private:
4404         IrisInstanceSemihosting* inst_semihost;
4405
4406     public:
4407         SemihostingManager(IrisInstanceSemihosting* inst_semihost_)
4408             : inst_semihost(inst_semihost_)
4409         {
4410         }
4411
4412         ~SemihostingManager()
4413         {
```

```
4414             // Interrupt any requests that are currently blocked
4415             unblock();
4416         }
4417 
4422         void enableExtensions()
4423         {
4424             inst_semihost->enableExtensions();
4425         }
4426 
4441         std::vector<uint8_t> readData(uint64_t fDes, size_t max_size = 0, uint64_t flags =
     semihost::DEFAULT)
4442         {
4443             return inst_semihost->readData(fDes, max_size, flags);
4444         }
4445 
4446         /*
4447          * @brief Write data for a given file descriptor
4448          *
4449          * @param   fDes        File descriptor to write to. Usually semihost::STDOUT or
     semihost::STDERR.
4450          * @param   data        Buffer containing the data to write.
4451          * @param   size        Size of the data buffer in bytes.
4452          * @return              Returns false if no client is registered for IRIS_SEMIHOSTING_OUTPUT
     events.
4453          */
4454         bool writeData(uint64_t fDes, const uint8_t* data, size_t size)
4455         {
4456             return inst_semihost->writeData(fDes, data, size);
4457         }
4458 
4459         /*
4460          * @brief Write data for a given file descriptor
4461          *
4462          * @param   fDes        File descriptor to write to. Usually semihost::STDOUT or
     semihost::STDERR.
4463          * @param   data        Buffer containing the data to write.
4464          * @return              Returns false if no client is registered for IRIS_SEMIHOSTING_OUTPUT
     events.
4465          */
4466         bool writeData(uint64_t fDes, const std::vector<uint8_t>& data)
4467         {
4468             return writeData(fDes, &data.front(), data.size());
4469         }
4470 
4485         std::pair<bool, uint64_t> semihostedCall(uint64_t operation, uint64_t parameter)
4486         {
4487             return inst_semihost->semihostedCall(operation, parameter);
4488         }
4489 
4490         /*
4491          * @brief Request premature exit from any blocking requests that are currently blocked.
4492          */
4493         void unblock()
4494         {
4495             return inst_semihost->unblock();
4496         }
4497     };
4498 
4506     SemihostingManager enableSemihostingAndGetManager()
4507     {
4508         inst_semihost.init();
4509         return SemihostingManager(inst_semihost);
4510     }
4511 
4515 };
4516 
4517 inline IrisInstanceBuilder::TableColumnBuilder IrisInstanceBuilder::TableBuilder::addColumn(const
     std::string& name)
4518 {
4519     // Add a new column with default info
4520     info.tableInfo.columns.resize(info.tableInfo.columns.size() + 1);
4521     TableColumnInfo& col = info.tableInfo.columns.back();
4522 
4523     col.name = name;
4524 
4525     return TableColumnBuilder(*this, col);
4526 }
4527 
4528 NAMESPACE_IRIS_END
4529 
4530 #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

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>
```

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.
- typedef IrisDelegate< uint64_t, const std::string &, MemoryReadResult &, uint64_t, uint64_t, std::vector< DisassemblyLine > & > iris::GetDisassemblyDelegate

  Get the disassembly of a chunk of memory.

9.25.1 Detailed Description

Disassembler add-on to IrisInstance.

Copyright

Copyright (C) 2016 Arm Limited. All rights reserved.

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
19 NAMESPACE_IRIS_START
20
21 class IrisInstance;
22 class IrisReceivedRequest;
23
39 typedef IrisDelegate<std::string&> GetCurrentDisassemblyModeDelegate;
40
51 typedef IrisDelegate<uint64_t, const std::string&, MemoryReadResult&,
52                      uint64_t, uint64_t, std::vector<DisassemblyLine>&>
53     GetDisassemblyDelegate;
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(GetDisassemblyDelegate delegate)
127     {
128         getDisassembly = 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     GetDisassemblyDelegate 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>
```

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) 2016-2021 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
24 NAMESPACE_IRIS_START
25
26 class IrisInstance;
27 class IrisReceivedRequest;
28
29 class EventStream;
30 class IrisEventRegistry;
31
44 typedef IrisDelegate<EventStream*&, const EventSourceInfo&, const std::vector<std::string>&>
    EventStreamCreateDelegate;
45
63 class IrisInstanceEvent
64 {
65 public:
66
67     /* ! What is a proxy event source?
68        - The event source in actual does not belong to this Iris instance, but instead belongs to another
    Iris instance (target).
69        - The event source is registered as a proxy in this Iris instance using Iris interface -
    event_registerProxyEventSource()
70        - This Iris instance acts as a proxy for those registered events.
71        - All interface calls (for example, eventStream_create) on the proxy event source are forwarded to
    the target instance.
72        - Similarly, all the created event streams in this Iris instance for the proxy event source are
    tagged as proxyForOtherInstance
73        - All the interface calls (for example, eventStream_enable) on such proxy event streams are
    forwarded to the target instance.
```

```
74        - Finally, the proxy event source can be deregistered using Iris interface -
      event_unregisterProxyEventSource()
75      */
76
80      struct ProxyEventInfo
81      {
82          InstanceId targetInstId{};      //target Iris instance Id
83          EventSourceId targetEvSrcId{};  //event source ID in target Iris instance
84          std::vector<EventStreamId> evStreamIds; //list of created event stream IDs
85          //Important note: When we create an event stream, we use the same esID for both - this and target
      Iris instance
86      };
87
91      struct EventSourceInfoAndDelegate
92      {
93          EventSourceInfo           info;
94          EventStreamCreateDelegate createEventStream;
95
96          bool isValid{true}; //deleteEventSource() sets isValid to false
97          bool isProxy{false};
98          ProxyEventInfo proxyEventInfo; //contains proper values only if isProxy=true
99      };
100
106     IrisInstanceEvent(IrisInstance* irisInstance = nullptr);
107     ~IrisInstanceEvent();
108
116     void attachTo(IrisInstance* irisInstance);
117
125     void setDefaultEsCreateDelegate(EventStreamCreateDelegate delegate);
126
139     EventSourceInfoAndDelegate& addEventSource(const std::string& name, bool isHidden = false);
140
148     uint64_t addEventSource(const EventSourceInfoAndDelegate& info);
149
158     EventSourceInfoAndDelegate& enhanceEventSource(const std::string& name);
159
165     void deleteEventSource(const std::string& eventName);
166
173     bool hasEventSource(const std::string& eventName);
174
182     const uint64_t *eventBufferGetSyncStepResponse(EventBufferId evBufId, RequestId requestId);
183
192     void eventBufferClear(EventBufferId evBufId);
193
201     bool isValidEvBufId(EventBufferId evBufId) const;
202
203 private:
204     // --- Iris function implementations ---
205
206     void impl_event_getEventSources(IrisReceivedRequest& request);
207
208     void impl_event_getEventSource(IrisReceivedRequest& request);
209
210     void impl_eventStream_create(IrisReceivedRequest& request);
211
212     void impl_eventStream_destroy(IrisReceivedRequest& request);
213
214     void impl_eventStream_enable(IrisReceivedRequest& request);
215
216     void impl_eventStream_disable(IrisReceivedRequest& request);
217
218     void impl_eventStream_getCounter(IrisReceivedRequest& request);
219
220     void impl_eventStream_setTraceRanges(IrisReceivedRequest& request);
221
222     void impl_eventStream_getState(IrisReceivedRequest& request);
223
224     void impl_eventStream_flush(IrisReceivedRequest& request);
225
226     void impl_eventStream_setOptions(IrisReceivedRequest& request);
227
228     void impl_eventStream_action(IrisReceivedRequest& request);
229
230     void impl_eventBuffer_create(IrisReceivedRequest& request);
231
232     void impl_eventBuffer_flush(IrisReceivedRequest& request);
233
234     void impl_eventBuffer_destroy(IrisReceivedRequest& request);
235
236     void impl_ec_eventBuffer(IrisReceivedRequest& request);
237
238     void register_ec_IRIS_INSTANCE_REGISTRY_CHANGED();
239     IrisErrorCode ec_IRIS_INSTANCE_REGISTRY_CHANGED(EventStreamId esId, const IrisValueMap& fields,
      uint64_t time,
240                                                     InstanceId sInstId, bool syncEc, std::string&
      errorMessageOut);
241
```

```
243
244     void impl_event_registerProxyEventSource(IrisReceivedRequest& request);
245
246     void impl_event_unregisterProxyEventSource(IrisReceivedRequest& request);
247
248     void impl_eventStream_create_proxy(IrisReceivedRequest& request);
249
250     IrisErrorCode impl_eventStream_destroy_target(IrisReceivedRequest& request, EventStream* evStream);
251
252     void impl_eventStream_enable_proxy(IrisReceivedRequest& request, EventStream* evStream);
253
254     void impl_eventStream_disable_proxy(IrisReceivedRequest& request, EventStream* evStream);
255
256     void impl_eventStream_getCounter_proxy(IrisReceivedRequest& request, EventStream* evStream);
257
258     void impl_eventStream_setTraceRanges_proxy(IrisReceivedRequest& request, EventStream* evStream);
259
260     void impl_eventStream_getState_proxy(IrisReceivedRequest& request, EventStream* evStream);
261
262     void impl_eventStream_flush_proxy(IrisReceivedRequest& request, EventStream* evStream);
263
264     void impl_eventStream_setOptions_proxy(IrisReceivedRequest& request, EventStream* evStream);
265
266     void impl_eventStream_action_proxy(IrisReceivedRequest& request, EventStream* evStream);
267
268     ProxyEventInfo& getProxyEventInfo(EventStream* evStream);
269
270     InstanceId getTargetInstId(EventStream* evStream);
271
273
274     EventStream* getEventStream(EventStreamId esId);
275
276     struct EventBufferStreamInfo;
277     struct EventBuffer;
278
281     const EventBufferStreamInfo* getEventBufferStreamInfo(InstanceId sInstId, EventStreamId esId) const;
282
284     EventBuffer* getEventBuffer(EventBufferId evBufId) const;
285
288     void eventBufferSend(EventBuffer *eventBuffer, bool flush);
289
291     void eventBufferDestroy(EventBufferId evBufId);
292
293     //Find a free event stream ID where a new EventStream can be added
294     //The returned ID is greater than or equal to 'minEsId'
295     EventStreamId findFreeEventStreamId(EventStreamId minEsId);
296
298
300     IrisInstance* irisInstance;
301
303     std::vector<EventSourceInfoAndDelegate> eventSources;
304
306     std::map<std::string, uint64_t>         srcNameToId;
307
310     std::vector<EventStream*> eventStreams;
311
314     std::vector<EventStreamId> freeEsIds;
315
317     EventStreamCreateDelegate defaultEsCreateDelegate;
318
320     IrisLogger log;
321
326     bool instance_registry_changed_registered{};
327
329     struct EventStreamOriginInfo
330     {
331         EventStreamId esId;
332         InstanceId sInstId;
333     };
334
336     struct EventBuffer
337     {
339         EventBuffer(const std::string& mode, uint64_t bufferSize, const std::string& ebcFunc, InstanceId
    ebcInstId, bool syncEbc, EventBufferId evBufId, IrisInstanceEvent *parent);
340
342         ~EventBuffer();
343
345         void clear();
346
348         const uint64_t* getResponse(RequestId requestId);
349
353         void getRequest(bool flush);
354
356         void addEventData(EventStreamInfoId esInfoId, uint64_t time, const uint64_t *fieldsU64Json);
357
359         void dropOldEvents(uint64_t targetBufferSizeU64);
360
```

```
362         std::string mode;
363
365         uint64_t bufferSizeU64{};
366
368         std::string ebcFunc;
369
371         InstanceId ebcInstId{IRIS_UINT64_MAX};
372
374         bool syncEbc{};
375
377         std::vector<EventStreamOriginInfo> eventStreams;
378
404         IrisU64JsonWriter writer;
405
407         uint64_t numEvents{};
408
410         size_t eventDataStartPos{};
411
414         IrisU64JsonWriter responseHeader;
415         size_t responseStartPos{};
416         size_t responseObjectPos{};
417         size_t responseArrayPos{};
418
421         IrisU64JsonWriter requestHeader;
422         size_t requestStartPos{};
423         size_t requestParamsPos{};
424         size_t requestReasonPos{};
425         size_t requestArrayPos{};
426
427         const uint64_t reasonSend  = 0x200000646E657304; // == "send"
428         const uint64_t reasonFlush = 0x20006873756C6605; // == "flush"
429
431         IrisInstanceEvent *parent{};
432     };
433     friend struct EventBuffer;
434
438     std::vector<EventBuffer*> eventBuffers;
439
442     std::vector<EventBufferId> freeEventBufferIds;
443
446     struct EventBufferStreamInfo
447     {
448         EventBuffer* eventBuffer{};
449         EventStreamInfoId esInfoId{};
450     };
451
459     std::vector<std::vector<EventBufferStreamInfo>> eventCallbackInfoToEventBufferStreamInfo;
460 };
461
467 class EventStream
468 {
469 public:
473     EventStream()
474         : enabled(false)
475         , req(nullptr)
476         , internal_req(nullptr)
477         , counter(false)
478         , isWaiting(false)
479         , selfReleaseAfterWaiting(false)
480     {
481     }
482
483     virtual ~EventStream() {}
484
496     void selfRelease()
497     {
498         // Disable the event stream if it is still enabled.
499         if (isEnabled())
500         {
501             disable();
502         }
503
504         // The request to destroy this event stream is nested and processed in the delegate to
505         // wait for the response, so it is not multi-threaded and no need to protect the variables.
506         if (!isWaiting)
507         {
508             delete this;
509             return;
510         }
511
512         // It is waiting for the response of the current request.
513         // Cancel the wait and release this object later (after the end of the wait).
514         req->cancel();
515         selfReleaseAfterWaiting = true;
516     }
517
528     virtual IrisErrorCode enable() = 0;
```

```
529
540     virtual IrisErrorCode disable() = 0;
541
551     virtual IrisErrorCode getState(IrisValueMap& fields)
552     {
553         (void) fields;
554         return E_not_supported_for_event_source;
555     }
556
566     virtual IrisErrorCode flush(RequestId requestId)
567     {
568         (void)requestId;
569         return E_not_supported_for_event_source;
570     }
571
589     virtual IrisErrorCode setOptions(const AttributeValueMap& options, bool eventStreamCreate,
        std::string& errorMessageOut)
590     {
591         (void)options;
592         (void)eventStreamCreate;
593         (void)errorMessageOut;
594
595         // Event streams which do not support options happily accept an empty options map.
596         return options.empty() ? E_ok : E_not_supported_for_event_source;
597     }
598
609     virtual IrisErrorCode action(const BreakpointAction& action_)
610     {
611         (void)action_;
612         return E_not_supported_for_event_source;
613     }
614
615     // Temporary: Keep PVModelLib happy. TODO: Remove.
616     virtual IrisErrorCode insertTrigger()
617     {
618         return E_not_supported_for_event_source;
619     }
620
621
622     // --- Functions for basic properties ---
623
639     void setProperties(IrisInstance* irisInstance, const EventSourceInfo* srcInfo,
640                        InstanceId ecInstId, const std::string& ecFunc, EventStreamId esId,
641                        bool syncEc);
642
648     bool isEnabled() const
649     {
650         return enabled;
651     }
652
658     EventStreamId getEsId() const
659     {
660         return esId;
661     }
662
668     const EventSourceInfo* getEventSourceInfo() const
669     {
670         return srcInfo;
671     }
672
679     InstanceId getEcInstId() const
680     {
681         return ecInstId;
682     }
683
684     // --- Functions for the counter mode ---
685
692     void setCounter(uint64_t startVal, const EventCounterMode& counterMode);
693
699     bool isCounter() const
700     {
701         return counter;
702     }
703
708     void setProxyForOtherInstance()
709     {
710         isProxyForOtherInstance = true;
711     }
712
718     bool IsProxyForOtherInstance() const
719     {
720         return isProxyForOtherInstance;
721     }
722
728     void setProxiedByInstanceId(InstanceId instId)
729     {
730         proxiedByInstanceId = instId;
```

```
731     }
732
738     bool IsProxiedByOtherInstance() const
739     {
740         return proxiedByInstanceId != IRIS_UINT64_MAX;
741     }
742
748     InstanceId getProxiedByInstanceId() const
749     {
750         return proxiedByInstanceId;
751     }
752
758     uint64_t getCountVal() const
759     {
760         return curVal;
761     }
762
763     // --- Functions for event stream with ranges
764
773     IrisErrorCode setRanges(const std::string& aspect, const std::vector<uint64_t>& ranges);
774
783     bool checkRangePc(uint64_t pc) const
784     {
785         return ranges.empty() || (aspect != ":pc") || checkRangesHelper(pc, ranges);
786     }
787
788     // --- Functions to emit the event callback ---
789     // Usage (example):
790     //     emitEventBegin(time, pc);    // Start to emit the callback.
791     //     addField(...);           // Add field value.
792     //     addField(...);           // Add field value.
793     //     ...
794     //     emitEventEnd();              // Emit the callback.
795
803     void emitEventBegin(IrisRequest& req, uint64_t time, uint64_t pc = IRIS_UINT64_MAX);
804
811     void emitEventBegin(uint64_t time, uint64_t pc = IRIS_UINT64_MAX);
812
822     void addField(const IrisU64StringConstant& field, uint64_t value)
823     {
824         addFieldRangeHelper(field, value);
825     }
826
836     void addField(const IrisU64StringConstant& field, int64_t value)
837     {
838         addFieldRangeHelper(field, value);
839     }
840
850     void addField(const IrisU64StringConstant& field, bool value)
851     {
852         addFieldRangeHelper(field, value);
853     }
854
864     template <class T>
865     void addField(const IrisU64StringConstant& field, const T& value)
866     {
867         fieldObj.member(field, value);
868     }
869
879     void addField(const IrisU64StringConstant& field, const uint8_t *data, size_t sizeInBytes)
880     {
881         fieldObj.member(field, data, sizeInBytes);
882     }
883
893     void addFieldSlow(const std::string& field, uint64_t value)
894     {
895         addFieldSlowRangeHelper(field, value);
896     }
897
907     void addFieldSlow(const std::string& field, int64_t value)
908     {
909         addFieldSlowRangeHelper(field, value);
910     }
911
921     void addFieldSlow(const std::string& field, bool value)
922     {
923         addFieldSlowRangeHelper(field, value);
924     }
925
935     template <class T>
936     void addFieldSlow(const std::string& field, const T& value)
937     {
938         fieldObj.memberSlow(field, value);
939     }
940
950     void addFieldSlow(const std::string& field, const uint8_t *data, size_t sizeInBytes)
951     {
```

```
952         fieldObj.memberSlow(field, data, sizeInBytes);
953     }
954
964     void emitEventEnd(bool send = true);
965
966 private:
968
972     bool counterTrigger();
973
975     bool checkRanges() const
976     {
977         return !aspectFound || checkRangesHelper(curAspectValue, ranges);
978     }
979
981     static bool checkRangesHelper(uint64_t value, const std::vector<uint64_t>& ranges);
982
984     template <typename T>
985     void addFieldRangeHelper(const IrisU64StringConstant& field, T value)
986     {
987         if (!aspect.empty() && aspect == toString(field))
988         {
989             aspectFound    = true;
990             curAspectValue = static_cast<uint64_t>(value);
991         }
992
993         fieldObj.member(field, value);
994     }
995
997     template <typename T>
998     void addFieldSlowRangeHelper(const std::string& field, T value)
999     {
1000         if (aspect == field)
1001         {
1002             aspectFound    = true;
1003             curAspectValue = static_cast<uint64_t>(value);
1004         }
1005
1006         fieldObj.memberSlow(field, value);
1007     }
1008
1009 protected:
1011
1013     IrisInstance* irisInstance;
1014
1016     const EventSourceInfo* srcInfo;
1017
1019     InstanceId ecInstId;
1020
1022     std::string ecFunc;
1023
1025     EventStreamId esId;
1026
1028     bool syncEc;
1029
1031     bool enabled;
1032
1034     IrisRequest*              req;
1035     IrisRequest*              internal_req;
1036     IrisU64JsonWriter::Object fieldObj;
1037
1039
1041     bool counter;
1042
1044     uint64_t startVal;
1045     uint64_t curVal;
1046
1048     EventCounterMode counterMode;
1049
1051
1052     std::string           aspect;
1053     std::vector<uint64_t> ranges;
1054
1056     bool aspectFound;
1057
1059     uint64_t curAspectValue;
1060
1062     bool isProxyForOtherInstance{false};
1063
1066     InstanceId proxiedByInstanceId{IRIS_UINT64_MAX};
1067
1068 private:
1071     bool isWaiting;
1072
1074     bool selfReleaseAfterWaiting;
1075 };
1076
1080 class IrisEventStream : public EventStream
```

```
1081 {
1082 public:
1083     IrisEventStream(IrisEventRegistry* registry_);
1084 
1085     virtual IrisErrorCode enable() IRIS_OVERRIDE;
1086 
1087     virtual IrisErrorCode disable() IRIS_OVERRIDE;
1088 
1089 private:
1090     IrisEventRegistry* registry;
1091 };
1092 
1096 class IrisEventRegistry
1097 {
1098 public:
1104     bool empty() const
1105     {
1106         return esSet.empty();
1107     }
1108 
1115     bool registerEventStream(EventStream* evStream);
1116 
1123     bool unregisterEventStream(EventStream* evStream);
1124 
1125     // --- Functions to emit the callback of all registered event streams ---
1126     // Usage (example):
1127     //     emitEventBegin(time, pc);    // Start to emit the callback.
1128     //     addField(...);           // Add field value.
1129     //     addField(...);           // Add field value.
1130     //     ...
1131     //     emitEventEnd();              // Emit the callback.
1132 
1133     void emitEventBegin(uint64_t time, uint64_t pc = IRIS_UINT64_MAX) const;
1134 
1145     template <class T>
1146     void addField(const IrisU64StringConstant& field, const T& value) const
1147     {
1148         for (std::set<EventStream*>::const_iterator i = esSet.begin(), e = esSet.end(); i != e; i++)
1149             (*i)->addField(field, value);
1150     }
1151 
1162     template <class T>
1163     void addFieldSlow(const std::string& field, const T& value) const
1164     {
1165         for (std::set<EventStream*>::const_iterator i = esSet.begin(), e = esSet.end(); i != e; i++)
1166             (*i)->addFieldSlow(field, value);
1167     }
1168 
1193     template <class T, typename F>
1194     void forEach(F && func) const
1195     {
1196         for (std::set<EventStream*>::const_iterator i = esSet.begin(), e = esSet.end(); i != e; i++)
1197         {
1198             T* t = static_cast<T*>(*i);
1199             func(*t);
1200         }
1201     }
1202 
1208     void emitEventEnd() const;
1209 
1210     typedef std::set<EventStream*>::const_iterator iterator;
1211 
1219     iterator begin() const
1220     {
1221         return esSet.begin();
1222     }
1223 
1231     iterator end() const
1232     {
1233         return esSet.end();
1234     }
1235 
1236     ~IrisEventRegistry()
1237     {
1238         // Disable any remaining event streams.
1239         // Calling disable() on an EventStream will cause esSet to be modified so we need to loop
       without
1240         // using iterators which become invalidated.
1241         while (!esSet.empty())
1242         {
1243             (*esSet.begin())->disable();
1244         }
1245     }
1246 
1247 private:
1248     // All registered event streams
1249     std::set<EventStream*> esSet;
```

```
1250 };
1251
1252 NAMESPACE_IRIS_END
1253
1254 #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 addBooleanParameter(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
145      IrisParameterBuilder addHiddenBooleanParameter(const std::string& name, const std::string&
        description)
146      {
147          ResourceInfo& info = addParameterInternal(parameter_prefix + name, 1, description, "numeric",
        true);
148
149          // Be explicit about the range even though there are only two possible values anyway.
150          info.parameterInfo.min.push_back(0);
151          info.parameterInfo.max.push_back(1);
152
153          // Add enum strings for the values
154          info.enums.push_back(EnumElementInfo(IrisValue(0), "false", ""));
155          info.enums.push_back(EnumElementInfo(IrisValue(1), "true", ""));
156
157          return IrisParameterBuilder(info);
158      }
159
166      const std::vector<ResourceInfo>& getParameterInfo() const
167      {
168          return parameters;
169      }
170
177      const std::vector<ResourceInfo>& getHiddenParameterInfo() const
178      {
179          return hidden_parameters;
180      }
181  };
182
183  NAMESPACE_IRIS_END
184
185  #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.

Copyright

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

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     // --- state ---
263
265     IrisInstance* irisInstance;
266
268     typedef std::vector<SpaceInfoAndAccess> SpaceInfoList;
269     SpaceInfoList                           spaceInfos;
270
272     typedef std::vector<AddressTranslationInfoAndAccess> SupportedTranslations;
273     SupportedTranslations                                supportedTranslations;
274
276     MemoryReadDelegate            memReadDelegate;
277     MemoryWriteDelegate           memWriteDelegate;
278     MemoryAddressTranslateDelegate translateDelegate;
279
282     MemoryGetSidebandInfoDelegate sidebandDelegate;
283
285     IrisLogger log;
286 };
287
288 NAMESPACE_IRIS_END
289
290 #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

Copyright (C) 2016 Arm Limited. All rights reserved.

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

Copyright (C) 2015-2019 Arm Limited. All rights reserved.

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 |= (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);
239
254     static void makeNamesHierarchical(std::vector<ResourceInfo>& resourceInfos);
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
194     IrisLogger log;
195
198     bool simulation_has_been_initialised;
199
201     std::vector<uint64_t> requests_waiting_for_instantiation;
202
204     unsigned logLevel{};
205
206 public:
214     IrisInstanceSimulation(IrisInstance*           iris_instance        = nullptr,
215                            IrisConnectionInterface* connection_interface = nullptr);
216     ~IrisInstanceSimulation();
217
223     void attachTo(IrisInstance* iris_instance);
224
230     void setConnectionInterface(IrisConnectionInterface* connection_interface_)
231     {
232         connection_interface = connection_interface_;
233     }
234
240     void setInstantiateDelegate(SimulationInstantiateDelegate delegate)
241     {
242         instantiate = delegate;
243     }
244
254     template <typename T, IrisErrorCode (T::*METHOD)(InstantiationResult&)>
255     void setInstantiateDelegate(T* instance)
256     {
257         setInstantiateDelegate(SimulationInstantiateDelegate::make<T, METHOD>(instance));
258     }
259
267     template <IrisErrorCode (*FUNC)(InstantiationResult&)>
268     void setInstantiateDelegate()
269     {
270         setInstantiateDelegate(SimulationInstantiateDelegate::make<FUNC>());
271     }
272
278     void setResetDelegate(SimulationResetDelegate delegate)
279     {
280         reset = delegate;
281     }
282
292     template <typename T, IrisErrorCode (T::*METHOD)(const IrisSimulationResetContext&)>
293     void setResetDelegate(T* instance)
294     {
295         setResetDelegate(SimulationResetDelegate::make<T, METHOD>(instance));
296     }
297
305     template <IrisErrorCode (*FUNC)(const IrisSimulationResetContext&)>
306     void setResetDelegate()
307     {
308         setResetDelegate(SimulationResetDelegate::make<FUNC>());
309     }
310
317     void setRequestShutdownDelegate(SimulationRequestShutdownDelegate delegate)
318     {
319         requestShutdown = delegate;
320     }
321
331     template <typename T, IrisErrorCode (T::*METHOD)()>
332     void setRequestShutdownDelegate(T* instance)
333     {
334         setRequestShutdownDelegate(SimulationRequestShutdownDelegate::make<T, METHOD>(instance));
335     }
336
344     template <IrisErrorCode (*FUNC)()>
345     void setRequestShutdownDelegate()
346     {
347         setRequestShutdownDelegate(SimulationRequestShutdownDelegate::make<FUNC>());
348     }
349
360     void setGetParameterInfoDelegate(SimulationGetParameterInfoDelegate delegate, bool cache_result =
    true)
361     {
362         getParameterInfo           = delegate;
363         parameter_info_cache_state = cache_result ? CACHE_EMPTY : CACHE_DISABLED;
364         cached_parameter_info.clear();
365     }
366
380     template <typename T, IrisErrorCode (T::*METHOD)(std::vector<ResourceInfo>&)>
```

```
381     void setGetParameterInfoDelegate(T* instance, bool cache_result = true)
382     {
383         typedef SimulationGetParameterInfoDelegate D;
384         setGetParameterInfoDelegate(D::make<T, METHOD>(instance), cache_result);
385     }
386
398     template <IrisErrorCode (*FUNC)(std::vector<ResourceInfo>&)>
399     void setGetParameterInfoDelegate(bool cache_result = true)
400     {
401         typedef SimulationGetParameterInfoDelegate D;
402         setGetParameterInfoDelegate(D::make<FUNC>(), cache_result);
403     }
404
411     void setSetParameterValueDelegate(SimulationSetParameterValueDelegate delegate)
412     {
413         setParameterValue = delegate;
414     }
415
425     template <typename T, IrisErrorCode (T::*METHOD)(const InstantiationParameterValue&)>
426     void setSetParameterValueDelegate(T* instance)
427     {
428         setSetParameterValueDelegate(SimulationSetParameterValueDelegate::make<T, METHOD>(instance));
429     }
430
438     template <IrisErrorCode (*FUNC)(const InstantiationParameterValue&)>
439     void setSetParameterValueDelegate()
440     {
441         setSetParameterValueDelegate(SimulationSetParameterValueDelegate::make<FUNC>());
442     }
443
452     void enterPostInstantiationPhase();
453
459     void setEventHandler(IrisInstanceEvent* handler);
460
467     void notifySimPhase(uint64_t time, IrisSimulationPhase phase);
468
480     void registerSimEventsOnGlobalInstance();
481
487     static std::string getSimulationPhaseName(IrisSimulationPhase phase);
488
494     static std::string getSimulationPhaseDescription(IrisSimulationPhase phase);
495
501     void setLogLevel(unsigned logLevel_) { logLevel = logLevel_; }
502
503 private:
505     void impl_simulation_getInstantiationParameterInfo(IrisReceivedRequest& request);
506
508     void impl_simulation_setInstantiationParameterValues(IrisReceivedRequest& request);
509
511     void impl_simulation_instantiate(IrisReceivedRequest& request);
512
514     void impl_simulation_reset(IrisReceivedRequest& request);
515
517     void impl_simulation_requestShutdown(IrisReceivedRequest& request);
518
520     void impl_simulation_waitForInstantiation(IrisReceivedRequest& request);
521
523     IrisErrorCode createEventStream(EventStream*& event_stream_out, const EventSourceInfo& info,
524                                     const std::vector<std::string>& fields);
525 };
526
527 NAMESPACE_IRIS_END
528
529 #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
128     EventBufferId evBufId{IRIS_UINT64_MAX};
129 };
130
131 NAMESPACE_IRIS_END
132
133 #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

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

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
```