ARM®

Model Trace Interface Reference Manual

1.1

Generated by Doxygen 1.3.9.1

ARM DUI 0819A

Contents

Chapter 1

Introduction to the Model Trace Interface

1.1 Overview

Using the Model Trace Interface Plugin Development Kit (MTIPDK) model integrators can create trace plugins, which are special shared libraries. These trace plugins can be loaded into Fast Models.

Fast Models provide trace in the form of a set of trace sources. While the simulation is running each trace source generates a stream of trace events. A trace event can be accompanied with a set of data fields. Each event of a single data source will always have the same set of data fields.

There are two possible actions that can be triggered by a trace event: A counter registered by the trace plugin can be increased, and a callback, also registered by the plugin, can be called. The callback can carry a subset of the data fields a trace source provides.

Callbacks and counters can be registered and unregistered during run-time of the simulation. However the simulation must be in stopped state.

1.2 The Model Trace Interface

The Model Trace Interface is a generic interface. The kind of data that should be traced is not represented in the interface functions. Instead the interface provides means to query the available trace sources at runtime, and dynamically pick at runtime a subset of the available data.

Also the data provided is dynamically described in a generic form. This allows for a trace plugin to adapt to new properties of a model, for example additional registers.

This approach was chosen to allow compatibility even with the Fast Models evolving.

1.3 Interface Versioning

Although the interface was designed to stay stable, it might be necessary in the future to change the interface. This is supported by the use of the CAInterface::ObtainInterface() paradigm of providing interfaces, and also by encapsulating classes in version namespaces, and the use of versioned header files.

To keep the path open to newer versions of the header files interface users must never refer to versioned headers. Instead the un-versioned wrapper versions of the header files must be used. This will also ensure to import versioned classes into the MTI namespace.

Examples: Must not use #include "MTI/ComponentTraceInterface_v1.h", use MTI/ComponentTraceInterface.h" instead. Must not use MTI::v1:ComponentTraceInterface, use MTI::ComponentTraceInterface.

1.4 Interface Walkthrough

This will be a quick run through the structure of the Model Trace Interface. It should give a first rough idea about how to use the interface.

1.4.1 GetCAInterface() - to be implemented by the trace plugin

The entry point of a trace plugin is the GetCAInterface() function, which returns a CAInterface pointer. The GetCAInterface() function must return a pointer to a class implementing the MTI::PluginFactory interface.

1.4.2 MTI::PluginFactory - to be implemented by the trace plugin

By implementation of the MTI::PluginFactory interface methods GetNumberOfParameters() and GetParameterInfos() a trace plugin can provide instantiation time parameters for the plugin. These will appear together with the instantiation time parameters of model, but prefixed by "TRACE" and the name of the trace plugin instance.

The implementation of the MTI::PluginFactory::Instantiate() method must return a pointer to a MTI::PluginInstance, again in form of a CAInterface pointer. The Instantiate method might be called multiple times if a user decides to create multiple instances of a trace plugin.

1.4.3 MTI::PluginInstance - to be implemented by the trace plugin

The method MTI::PluginInstance::RegisterSimulation(CAInterface ∗) will be called once for every instance of the plugin at model instantiation time. It's task is to query available trace sources of the system, and to register those trace sources which the plugin is interested in.

Using the ObtainInterface() paradigm the CAInterface pointer can be converted into a SystemTraceInterface pointer.

1.4.4 MTI::SystemTraceInterface - provided by model

The SystemTraceInterface provides system-wide access to trace data. A trace plugin can decide to use trace data system level, or to focus on a subsystem or a single component providing trace data.

The interface provides methods to query the list of components that contain trace support. Currently these will be all the cores of a system, but in the future this might also include peripherals and other components providing trace data. Use of one of the GetComponentTrace() methods will return a CAInterface pointer, which by using ObtainInterface() can be converted into a MTI::ComponentTrace pointer.

1.4.5 MTI::ComponentTrace - provided by model

Using the ComponentTrace interface the trace plugin can access all trace event sources (pointer to TraceSource) that a component provides. It contains methods to enumerate all trace sources, and a method to query a trace source with a known name.

Also methods exist to query what kind of component is providing the trace, and which version the component has.

1.4.6 MTI::TraceSource - provided by the model

The TraceSource class is the heart of the Model Trace Interface. It contains methods to query the number and form of the data fields that a event of the trace source might provide (via the MTI::EventFieldType class). It provides the MTI::TraceSource::RegisterCounter() method which can be used to count events of a trace source. If also provides the CreateEventClass() method which is used to select a subset of the data fields using a bitmask. The resulting MTI::EventClass is required to register a callback with a trace source. Multiple event classes can be registered with a trace source.

1.4.7 MTI::EventFieldType - provided by the model

This class is a single data field of a trace source. It defines a type and a size for the data. In case that the type of a field is an enumeration (MTI_ENUM) it also provides methods to query the enumeration constants, allowing a fully generic handling of trace data, if required.

1.4.8 MTI::EventClass - provided by the model

An EventClass represents a selection of a subset of data fields of a trace source. It is used to register a callback using the RegisterCallback() method (UnregisterCallback() to remove it). Multiple callbacks can be registered.

Callbacks bring a pointer to an MTI::EventRecord structure with them. Since the subset of data traced is only fixed at run-time, the layout of the EventRecord is also run-time defined. There are some methods to query the dynamical layout of the EventRecord.

If a trace source is switched on in the middle of a simulation run often the current state of the model is unknown. For example the current values of the register file might be unknown, and only registers updated will trigger a trace event. For this reason some TracesSources implement the DumpState() method which will trigger callbacks to report the entire internal state. Another use-case could be that a trace plugin is interested in the value of registers only at certain points. It could then call DumpState() to be updated only at those points.

1.4.9 MTI::EventRecord - provided by the model

A pointer to an EventRecord is one of the three arguments passed to a callback call. Due to the dynamic layout of event records the EventRecord structure contains no data members except for the event_class_id. Other data members can be accessed via helper methods.

Chapter 2

Class Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

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

Chapter 5

Class Documentation

5.1 CAInterface Class Reference

```
#include <CAInterface.h>
```

Public Member Functions

- virtual CAInterface ∗ ObtainInterface (if_name_t ifName, if_rev_t minRev, if_rev_t ∗actualRev)=0
- template<class TARGET_IF >
  TARGET_IF ∗ ObtainPointer ()

Static Public Member Functions

- static if_name_t IFNAME ()
- static if_rev_t IFREVISION ()

5.1.1 Detailed Description

Base class for extensible component interfaces.

CAInterface provides a basis for a software model built around 'components' and 'interfaces'.

An 'interface' is an abstract class (consisting entirely of pure virtual methods), which derives from CAInterface, and which provides a number of methods for interacting with a component.

A 'component' is a black-box entity that has a unique identity. A component provides concrete implementations of one or more interfaces. Each of these interfaces may expose different facets of the component's behaviour. These interfaces are the only way to interact with the component.

There is no way for a client to enumerate the set of interfaces that a component implements; instead, the client must ask for specific interfaces by name. If the component doesn't implement the requested interface, it returns a NULL pointer.

(The implementation of a component's interfaces may be provided by one or several interacting C++ objects; this is an impementation detail that is transparent to the client).

Interfaces are identified by a string name (of type if_name_t), and an integer revision (type if_rev_t). A higher revision number indicates a newer revision of the same interface.

class CAInterface is the base class for all interfaces. It defines a method, CAInterface::ObtainInterface(), that allows a client to obtain a reference to any of the interfaces that the component implements.

The client specifies the id and revision of the interface that it wants to request. The component can return NULL if it doesn't implement that interface, or only implements a lower revision.

Since each interface derives from CAInterface, a client can call ObtainInterface() on any one interface pointer to obtain a pointer to any other interface implemented by the same component.

The following rules govern the use of components and interfaces.

Each component is distinct. No two components can return the same pointer for a given interface. An ObtainInterface() call on one component must not return an interface on a different component.

Each interface consists of a name, a revision number, and a C++ abstract class definition. The return value of ObtainInterface() is either NULL, or is a pointer that can be cast to the class type.

Where two interfaces have the same if_name_t, the newer revision of the interface must be backwards-compatible with the old revision. (This includes the binary layout of any data-structures that it uses, and the semantics of any methods).

During the lifetime of a component, any calls to ObtainInterface() for a given interface name and revision must always return the same pointer value. It must not matter which of the component's interfaces is used to invoke ObtainInterface(). ∗ All components must implement the interface named 'eslapi::CAInterface' revision 0, which implements class eslapi::CAInterface.

5.1.2 Member Function Documentation

5.1.2.1 static if_name_t IFNAME ( ) `[inline, static]`

Static helper method to return the interface name for the CAInterface.

Reimplemented in ComponentTraceInterface, ParameterInterface, PluginFactory, PluginFactory, PluginInstance, and SystemTraceInterface.

5.1.2.2 static if_rev_t IFREVISION ( ) `[inline, static]`

Static helper method to return the current interface revision for CAInterface.

Reimplemented in ComponentTraceInterface, ParameterInterface, PluginFactory, PluginFactory, PluginInstance, and SystemTraceInterface.

5.1.2.3 virtual CAInterface∗ ObtainInterface (if_name_t *ifName*, if_rev_t *minRev*, if_rev_t ∗ *actualRev*) `[pure virtual]`

Ask the component for a handle to a different interface, if the component implements that interface. The return value is NULL if the component doesn't implement the specified interface; otherwise it is a pointer to an instance of CAInterface that can be cast to the appropriate interface type.

Parameters

`in`	*ifName*	Name identifying the requested interface.
`in`	*minRev*	Specify the minimum minor revision required.
`out`	*actualRev*	If not NULL, used to returns the actual revision number implemented.

Returns

Pointer to the requested interface, or NULL.

5.1.2.4 TARGET_IF∗ ObtainPointer ( ) `[inline]`

Helper method to make it easier to get an interface pointer.

This returns a pointer of type TARGET_IF∗, obtained by calling ObtainInterface() on this component. If this component doesn't provide a compatible version of the desired interface, it returns a NULL pointer.

The desired interface and version must be specified within the TARGET_IF class definition by static helper methods IFNAME() and IFREVISION() -- following the examples provided above.

This method is not technically part of the interface (ABI) exposed by a component; it is just defined here to make it easier to use the interface from C++ code.

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

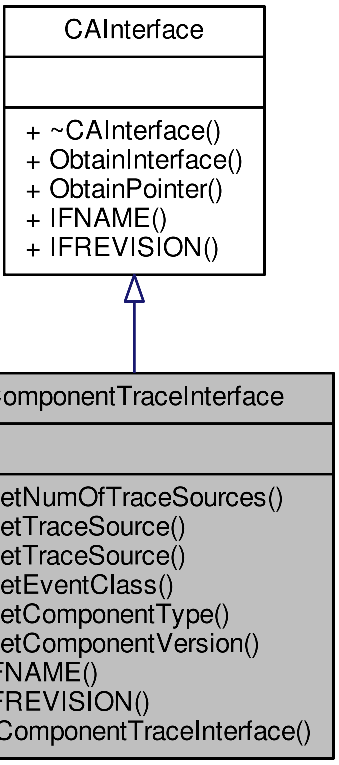
- CAInterface.h

5.2 ComponentTraceInterface Class Reference

Class to access all the trace sources of a component.

`#include <ComponentTraceInterface_v1.h>`

Collaboration diagram for ComponentTraceInterface:



Public Member Functions

- virtual SourceIndex GetNumOfTraceSources () const =0
- virtual TraceSource ∗ GetTraceSource (SourceIndex index) const =0
- virtual TraceSource ∗ GetTraceSource (const char ∗name) const =0
- virtual const EventClass ∗ GetEventClass (EventClassId event_class_id) const =0
- virtual const char ∗ GetComponentType () const =0
- virtual const char ∗ GetComponentVersion () const =0

Static Public Member Functions

- static eslapi::if_name_t IFNAME ()
- static eslapi::if_rev_t IFREVISION ()

Protected Member Functions

- virtual ∼ComponentTraceInterface ()

5.2.1 Detailed Description

Class to access all the trace sources of a component.

This class contains methods to query the trace sources a single component provides. It is also possible to directly access a trace source of known name.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 virtual ~ComponentTraceInterface ( ) `[inline, protected, virtual]`

Not to be called by interface users.

An interface user must never delete an instance of type ComponentTraceInterface as returned by SystemTraceInterface::GetComponentTrace(). ComponentTraceInterface instances live as long as their simulation.

5.2.3 Member Function Documentation

5.2.3.1 static eslapi::if_name_t IFNAME ( ) `[inline, static]`

Static helper method to return the interface name for the CAInterface.

Reimplemented from CAInterface.

5.2.3.2 static eslapi::if_rev_t IFREVISION ( ) `[inline, static]`

Static helper method to return the current interface revision for CAInterface.

Reimplemented from CAInterface.

5.2.3.3 virtual SourceIndex GetNumOfTraceSources ( ) const `[pure virtual]`

Return the number of trace sources of the component.

5.2.3.4 virtual TraceSource* GetTraceSource (SourceIndex *index*) const `[pure virtual]`

Return a pointer to a trace source by index.

Parameters

index	Index should be 0 to GetNumOfTraceSources()-1.

Returns

Returns a pointer to a trace source. Will return 0 if an illegal index is given.

5.2.3.5 virtual TraceSource* GetTraceSource (const char * *name*) const `[pure virtual]`

Return a pointer to a trace source by name.

Parameters

name	Name of a trace source. Trace source names are case sensitive.

Returns

Returns a pointer to a trace source. Will return 0 if no trace source with the specified name exists.

5.2.3.6 virtual const EventClass∗ GetEventClass (EventClassId *event_class_id*) const `[pure virtual]`

Get the event class with the unique event class ID.

Parameters

event_class_id	The ID for which the EventClass pointer should be returned.

Returns

Returns a pointer to a EventClass. The instance returned lives as long as the simulation and must not be freed. If no events exist with the specified ID a 0 pointer is returned.

5.2.3.7 virtual const char∗ GetComponentType ( ) const `[pure virtual]`

Get the type of a trace component. This will help a trace plugin to identify what kind of component is providing trace. Examples of strings returned are "ARM1176JZF-S" or "ARM_Cortex-A8".

Returns

Returns a short typename identifying the kind of component. A component which does not implement this functionality should return 0.

5.2.3.8 virtual const char∗ GetComponentVersion ( ) const `[pure virtual]`

This will return a string identifying the version of the component providing trace. The format of the version string is up to the component implementor. However components provided as part of Fast Models will use the number of the Fast Models release as the version string (for example "5.1.24").

Returns

A string giving the component's version. A component which does not implement this functionality should return 0.

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

- ComponentTraceInterface_v1.h

5.3 EventClass Class Reference

```
#include <ComponentTraceInterface_v1.h>
```

Public Member Functions

- virtual EventClassId GetId () const =0
- virtual const TraceSource ∗ GetSource () const =0
- virtual FieldMask GetMask () const =0
- virtual size_t GetEventRecordSize () const =0
- virtual ValueIndex GetNumValues () const =0
- virtual const EventFieldType ∗ GetEventField (ValueIndex vidx) const =0
- virtual ValueIndex GetValueIndex (const char ∗name) const =0
- virtual size_t GetValueOffset (ValueIndex vidx) const =0
- virtual Status RegisterCallback (CallbackT callback, void ∗user_ptr=0)=0
- virtual Status UnregisterCallback (CallbackT callback, void ∗user_ptr=0)=0
- virtual Status DumpState (CallbackT callback, void ∗user_ptr=0)=0

Protected Member Functions

- virtual ∼EventClass ()

5.3.1 Detailed Description

An event class describes a subset of fields selected from a certain trace source. Once an event class has been created (with TraceSource::CreateEventClass()) callbacks can be registered with it.

Callbacks registered with the EventClass are passed an EventRecord which contains the values of the fields requested. EventClass includes methods which allow the caller to determine the size and layout of the EventRecord.

Every EventClass has a unique EventClassId. This EventClassId is also included in the EventRecord. This makes it possible to find the EventClass and TraceSource that the EventRecord belongs to.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 virtual ∼EventClass ( ) `[inline, protected, virtual]`

Not to be called by an interface user.

An interface user must never delete an instance of type EventClass. It is created by a call to Trace-Source::CreateEventClass() and will live as long as the simulation.

5.3.3 Member Function Documentation

5.3.3.1 virtual EventClassId GetId ( ) const `[pure virtual]`

Return the unique EventClassId of this EventClass.

5.3.3.2 virtual const TraceSource∗ GetSource ( ) const `[pure virtual]`

Return the TraceSource this EventClass is defined for.

5.3.3.3 virtual FieldMask GetMask ( ) const `[pure virtual]`

Get the mask that was used for the creation of this EventClass.

5.3.3.4 virtual size_t GetEventRecordSize ( ) const `[pure virtual]`

Return the size of an EventRecord produced by this EventClass. If the EventClass contains variable sized fields this is the maximum size of EventRecord that might be generated.

5.3.3.5 virtual ValueIndex GetNumValues ( ) const `[pure virtual]`

Return the number of values in the EventRecord produced. This is equal to the number of bits set in the FieldMask used to produce this EventClass.

5.3.3.6 virtual const EventFieldType* GetEventField (ValueIndex *vidx*) const `[pure virtual]`

Returns the EventFieldType describing the value of the EventRecord produced by this EventClass.

5.3.3.7 virtual ValueIndex GetValueIndex (const char * *name*) const `[pure virtual]`

Return the index of the value by name.

Parameters

name	The name of the value. Value names are the same as the field name they are selected from.

Returns

The index of the value in the range 0 to GetNumValues()-1. If no value with the specified name exists then ValueIndex(-1) is returned.

5.3.3.8 virtual size_t GetValueOffset (ValueIndex *vidx*) const `[pure virtual]`

Return the offset in bytes of a specified value inside the EventRecord.

Parameters

vidx	A value index in the range 0 to GetNumValues()-1.

Returns

Returns the offset in bytes. When an illegal value index is given 0 is returned. (0 is the offset of the event class id, so can never be a legal offset of a value.)

5.3.3.9 virtual Status RegisterCallback (CallbackT *callback*, void * *user_ptr* = 0) `[pure virtual]`

Register a callback with this EventClass. Every time this trace event happens the callback function is called. Depending on the event the callback may be called with a high frequency. In this case it is important to make the callback function execute quickly to keep the impact on the simulation performance low. If more complex operations are required then one might create a separate worker thread to do the expensive operations. The callback function would then just pass a short signal to the worker thread and return to the simulation.

It is possible to register multiple callbacks for the same EventClass. All of the callbacks will be called when an event occurs. The order in which they are called is undefined.

The same callback can be registered multiple times with different values of user_ptr. However the same callback/user_ptr combination should not be registered more than once, otherwise the behavior is undefined.

The simulation must be in the stopped state to register (or unregister) callbacks.

Parameters

callback	This is a pointer to a user defined "C" function (can be a static class member function).
user_ptr	The use of this pointer is left to the interface user. It is passed as a parameter when the callback is called. A typical use case is to pass a class pointer which can then be used in a wrapper function to call a class member function.

Returns

Returns MTI_OK if the callback was successfully registered.

5.3.3.10 virtual Status UnregisterCallback (CallbackT *callback*, void ∗ *user_ptr* = 0) `[pure virtual]`

Unregister a callback registered earlier by RegisterCallback(). The simulation must be in the halted state to unregister (or register) callbacks.

Parameters

callback	The pointer earlier passed to RegisterCallback().
user_ptr	The user_ptr used to register the callback. The combination of callback and user_ptr uniquely identifies the registration which should be removed.

Returns

Returns MTI_OK if the callback was successfully removed.

5.3.3.11 virtual Status DumpState (CallbackT *callback*, void ∗ *user_ptr* = 0) `[pure virtual]`

Dump the current state of a trace source through the callback provided. This could be useful if the current internal state of a trace source is unknown, for example after switching on tracing in the middle of a simulation run. A typical example would be a trace source describing register events. Since only register writes trigger a normal callback the status of the register file is unknown when tracing is started. A call to DumpState will cause a callback for every register of the register file, with the current register value.

Not all trace sources will implement this functionality.

The simulation does not have to be stopped in order for this call to work. It is permitted to call this from within a another callback, provided that that callback is from a different TraceSource.

Parameters

callback	The callback function the callbacks should be made to.
user_ptr	A pointer passed through to the callback, with user defined content.

Returns

Returns MTI_OK if the call was successful. The function will return only after the callbacks have been made.

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

- ComponentTraceInterface_v1.h

5.4 EventFieldType Class Reference

```
#include <ComponentTraceInterface_v1.h>
```

Public Types

- enum Type {
  MTI_UNSIGNED_INT, **MTI_SIGNED_INT**, **MTI_BOOL**, **MTI_ENUM**,
  MTI_FLOAT, **MTI_STRING** }
- typedef uint16_t Size
- typedef uint32_t EnumIndex
- typedef std::pair< uint32_t, const char ∗ > EnumConstant

Public Member Functions

- virtual const TraceSource ∗ GetSource () const =0
- virtual FieldIndex GetIndex () const =0
- virtual const char ∗ GetName () const =0
- virtual const char ∗ GetDescription () const =0
- virtual Type GetType () const =0
- virtual Size GetSize () const =0
- virtual Size GetMaxSize () const =0
- virtual EnumIndex GetNumOfEnumConstants () const =0
- virtual const EnumConstant GetEnumConstant (EnumIndex enum_index) const =0
- virtual const char ∗ LookupEnum (uint32_t value) const =0

Protected Member Functions

- virtual ∼EventFieldType ()

5.4.1 Detailed Description

This class describes the field of an event emitted by a trace source.

5.4.2 Member Typedef Documentation

5.4.2.1 typedef uint32_t EnumIndex

This type is used to enumerate enum constants for EventFieldType of type MTI_ENUM.

5.4.2.2 typedef std::pair<uint32_t, const char ∗> EnumConstant

This type is used to return one enum constant by the GetEnumConstant() method.

5.4.3 Member Enumeration Documentation

5.4.3.1 enum Type

This describes the type of a field of a trace event.

5.4.4 Constructor & Destructor Documentation

5.4.4.1 virtual ∼EventFieldType ( ) `[inline, protected, virtual]`

An interface user must never delete an instance of type EventFieldType as returned by EventClass::GetField(). The instances live as long as the simulation they belong to.

5.4.5 Member Function Documentation

5.4.5.1 virtual FieldIndex GetIndex ( ) const `[pure virtual]`

Get the index of this event field type. Event fields are numbered from 0 to TraceSource::GetNumFields()-1.

5.4.5.2 virtual Size GetSize ( ) const `[pure virtual]`

Return the size of the data of this event field in bytes. A size of 0 indicates a field of variable size. The maximum size in this case is given by GetMaxSize(). The actual size can be acquired via EventRecord::GetSize().

5.4.5.3 virtual Size GetMaxSize ( ) const `[pure virtual]`

Return the maximum size of this field. For fields of fixed size this will return the same as GetSize().

5.4.5.4 virtual EnumIndex GetNumOfEnumConstants ( ) const `[pure virtual]`

For fields of type MTI_ENUM this will return how many enumeration constants exist.

5.4.5.5 virtual const EnumConstant GetEnumConstant (EnumIndex *enum_index*) const `[pure virtual]`

For a field of type MTI_ENUM this can return one enumeration constant.

Parameters

enum_index	The enumeration index must be in the range 0 to GetNumOfEnumConstants()-1. It is NOT the 'value' of the enum - use the LookupEnum() method to look up single enum values.

Returns

This returns a pair<uint32_t, const char ∗>. If the enum_index is outside the range 0 .. GetNumOfEnumConstants()-1 then a pair with a 0 const char pointer is returned.

5.4.5.6 virtual const char∗ LookupEnum (uint32_t *value*) const `[pure virtual]`

Look up a single enumeration constant.

Parameters

value	The value of the enum constant to look up.

Returns

A pointer to a string constant is returned. If no enum constant exists for the value given then 0 is returned.

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

- ComponentTraceInterface_v1.h

5.5 EventRecord Struct Reference

`#include <ComponentTraceInterface_v1.h>`

Public Member Functions

- template<typename T >
  T Get (const EventClass ∗event_class, ValueIndex vidx) const
- bool GetBool (const EventClass ∗event_class, ValueIndex vidx) const
- const uint8_t ∗ GetPtr (const EventClass ∗event_class, ValueIndex vidx) const
- EventFieldType::Size GetVariableSize (const EventClass ∗event_class, ValueIndex vidx) const
- EventFieldType::Size GetSize (const EventClass ∗event_class, ValueIndex vidx) const
- template<typename T >
  T GetAs (const EventClass ∗event_class, ValueIndex vidx) const

Public Attributes

- EventClassId event_class_id

5.5.1 Detailed Description

An EventRecord is the structure passed when calling callback functions. Except for the event_class_id field the layout of the data is dynamic and depends on the fields selected when creating an EventClass. Accessor methods are used to access the data. This allows the binary layout of the event to change without requiring MTI plugins to be recompiled.

5.5.2 Member Function Documentation

5.5.2.1 T Get (const EventClass ∗ *event_class,* ValueIndex *vidx*) const [inline]

This template method is used to access a value of known type and size. Since the type and size of a event field might change with different models it is recommended to at least once check the size and type of a value before using this method. Alternatively the GetAs() method should be used. Types permitted are uint8_t, uint16_t, uint32_t, uint64_t, int8_t, int16_t, int32_t and int64_t.

5.5.2.2 bool GetBool (const EventClass ∗ *event_class,* ValueIndex *vidx*) const [inline]

This is a special method to access boolean values of an EventRecord.

5.5.2.3 const uint8_t∗ GetPtr (const EventClass ∗ *event_class,* ValueIndex *vidx*) const [inline]

This method helps to access values not accessible via the Get<> or GetAs<> methods. These are for example values of variable size, or those which have a size different from a C type (not 8, 16, 32 or 64 bits). The method returns a pointer to the first byte of the data.

5.5.2.4 EventFieldType::Size GetVariableSize (const EventClass ∗ *event_class,* ValueIndex *vidx*) const [inline]

This method returns the size of a value of an EventRecord in bytes. The combination of GetVariable-Size()/GetPtr() is used to access variable sized values. It must not be used on values of fixed size (in that case use GetSize() instead).

5.5.2.5 EventFieldType::Size GetSize (const EventClass ∗ *event_class,* ValueIndex *vidx*) const [inline]

This method returns the size of a value. It can be used on both fixed and variable sized fields.

5.5.2.6 T GetAs (const EventClass ∗ *event_class,* ValueIndex *vidx*) const [inline]

This convenience method will access a value of an EventRecord, and convert it into the specified data type Depending on the size of the EventRecord's value and the target type, the data might be extended or truncated. Signed values will be correctly sign-extended if necessary.

5.5.3 Member Data Documentation

5.5.3.1 EventClassId event_class_id

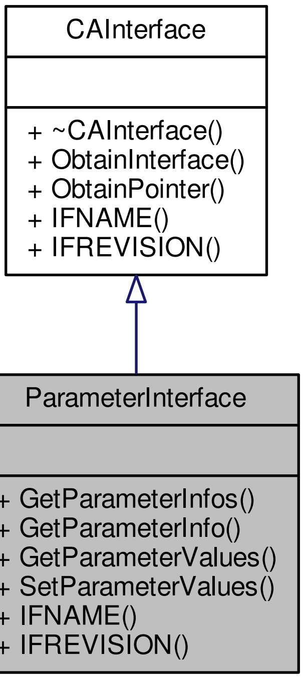
The event_class_id can be used to look up the EventClass this EventRecord belongs to. Since callbacks are always passed a pointer to the EventClass it isn't generally necessary to use this inside callbacks.

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

- ComponentTraceInterface_v1.h

5.6 ParameterInterface Class Reference

Collaboration diagram for ParameterInterface:



Public Member Functions

- virtual eslapi::CADIReturn_t GetParameterInfos (uint32_t startIndex, uint32_t desiredNumOfParams, uint32_t ∗actualNumOfParams, eslapi::CADIParameterInfo_t ∗params)=0
- virtual eslapi::CADIReturn_t GetParameterInfo (const char ∗parameterName, eslapi::CADIParameterInfo_-t ∗param)=0
- virtual eslapi::CADIReturn_t GetParameterValues (uint32_t parameterCount, uint32_t ∗actualNumOfParamsRead, eslapi::CADIParameterValue_t ∗paramValuesOut)=0
- virtual eslapi::CADIReturn_t SetParameterValues (uint32_t parameterCount, eslapi::CADIParameterValue_-t ∗parameters, eslapi::CADIFactoryErrorMessage_t ∗error)=0

Static Public Member Functions

- static eslapi::if_name_t IFNAME ()
- static eslapi::if_rev_t IFREVISION ()

5.6.1 Member Function Documentation

5.6.1.1 static eslapi::if_name_t IFNAME ( ) `[inline, static]`

Static helper method to return the interface name for the CAInterface.

Reimplemented from CAInterface.

5.6.1.2 static eslapi::if_rev_t IFREVISION ( ) `[inline, static]`

Static helper method to return the current interface revision for the CAInterface.

Reimplemented from CAInterface.

5.6.1.3 virtual eslapi::CADIReturn_t GetParameterInfos (uint32_t *startIndex*, uint32_t *desiredNumOfParams*, uint32_t ∗ *actualNumOfParams*, eslapi::CADIParameterInfo_t ∗ *params*) `[pure virtual]`

Get list of supported parameters and parameter information.

Parameters

`in`	*startIndex*	The index of the first parameter. Not a parameter id !
`in`	*desiredNumOf-Params*	The maximum number of parameters this call should return.
`out`	*actualNumOf-Params*	The number of parameter infos returned by the call. If this is smaller than desiredNumOfParams and the return value is CADI_STATUS_OK then there end of the parameter list has been reached. If this is equal to desiredNumOfParams there are more parameter infos for which another call should be made, likely with startIndex increased by actualNumOfParams.
`out`	*params*	A field that will contain the parameter infos after the call. The array is provided by the caller and must be large enough to contain desiredNumOfParams parameter infos.

5.6.1.4 virtual eslapi::CADIReturn_t GetParameterInfo (const char ∗ *parameterName*, eslapi::CADIParameterInfo_t ∗ *param*) `[pure virtual]`

Get parameter info for a specific parameter name.

Parameters

`in`	*parameter-Name*	Name of the parameter to be retrieved. This is the 'local' name in the model, not the global hierarchical name.
`out`	*param*	Points to a single CADIParameterInfo_t buffer which should be preinitialized by the caller and filled with data by the callee.

5.6.1.5 virtual eslapi::CADIReturn_t GetParameterValues (uint32_t *parameterCount*, uint32_t ∗ *actualNumOfParamsRead*, eslapi::CADIParameterValue_t ∗ *paramValuesOut*) `[pure virtual]`

Get current parameter values.

Parameters

in	*parameter-Count*	Length of array paramValuesOut.
out	*actualNumOf-ParamsRead*	Number of valid entries in paramValuesOut. Should be initialized to 0 by caller. If an error is returned and this is $>$ 0 then the first actualNumOfParams entries are valid and caused no error. In this case the entry paramValuesOut[actualNumOfParamsRead] caused the error.
out	*paramValue-sOut*	Output buffer of parameter values.

5.6.1.6 virtual eslapi::CADIReturn_t SetParameterValues (uint32_t *parameterCount*, eslapi::CADIParameterValue_t ∗ *parameters*, eslapi::CADIFactoryErrorMessage_t ∗ *error*) `[pure virtual]`

Set the parameters to new values.

Parameters

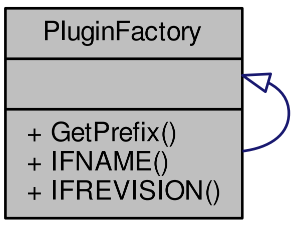
in	*parameter-Count*	Number of parameters for which a new value should be set. Can be a subset of the parameters reported via GetParameterInfos.
	parameters	An error of parameter values. Allocated by the called, must contain as many entries as parameterCount specifies.
out	*error*	Can be used to return an error message.

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

- ParameterInterface_v0.h

5.7 PluginFactory Class Reference

Collaboration diagram for PluginFactory:



Public Member Functions

- virtual const char ∗ GetPrefix ()

Static Public Member Functions

- static eslapi::if_name_t IFNAME ()
- static eslapi::if_rev_t IFREVISION ()

5.7.1 Member Function Documentation

5.7.1.1 static eslapi::if_name_t IFNAME ( ) `[inline, static]`

Static helper method to return the interface name for the CAInterface.

Reimplemented from PluginFactory.

5.7.1.2 static eslapi::if_rev_t IFREVISION ( ) `[inline, static]`

Static helper method to return the current interface revision for CAInterface.

Reimplemented from PluginFactory.

5.7.1.3 virtual const char* GetPrefix ( ) `[inline, virtual]`

Return the name if this plugin prefix.

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

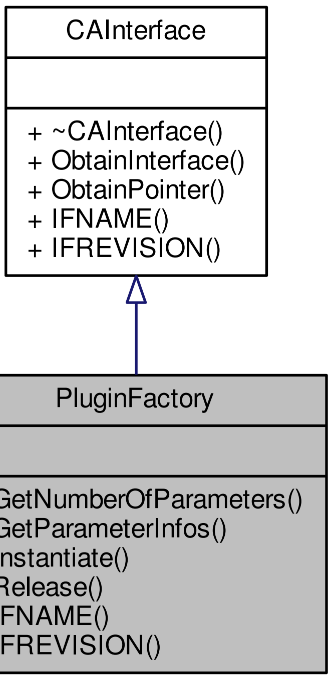
- PluginFactory_v1.h

5.8 PluginFactory Class Reference

Class providing a method to instantiate a PluginInstance.

`#include <PluginFactory_v0.h>`

Collaboration diagram for PluginFactory:



Public Member Functions

- virtual uint32_t GetNumberOfParameters ()=0
- virtual eslapi::CADIReturn_t GetParameterInfos (eslapi::CADIParameterInfo_t ∗parameter_info_-list)=0
- virtual CAInterface ∗ Instantiate (const char ∗instance_name, uint32_t number_of_parameters, eslapi::CADIParameterVal
  t ∗parameter_values)=0
- virtual void Release ()=0

Static Public Member Functions

- static eslapi::if_name_t IFNAME ()
- static eslapi::if_rev_t IFREVISION ()

5.8.1 Detailed Description

Class providing a method to instantiate a PluginInstance.

This interface must be implemented by anybody implementing a (trace) plugin. The interface provides methods to define instantiation time parameters and create an instance of a trace plugin. An implementation of the class must implement an CAInterface::ObtainInterface() method which in will usually return a pointer to itself if an IFNAME() interface is requested.

5.8.2 Member Function Documentation

5.8.2.1 static eslapi::if_name_t IFNAME ( ) [inline, static]

Static helper method to return the interface name for the CAInterface.

Reimplemented from CAInterface.

Reimplemented in PluginFactory.

5.8.2.2 static eslapi::if_rev_t IFREVISION ( ) [inline, static]

Static helper method to return the current interface revision for CAInterface.

Reimplemented from CAInterface.

Reimplemented in PluginFactory.

5.8.2.3 virtual uint32_t GetNumberOfParameters ( ) [pure virtual]

This should return the number of instantiation time parameters. Return 0 if a plugin does not have parameters.

5.8.2.4 virtual eslapi::CADIReturn_t GetParameterInfos (eslapi::CADIParameterInfo_t ∗ *parameter_info_list*) [pure virtual]

Return the instantiation time parameters. The caller must provide an array of length GetNumberOfParameters() of CADIParameterInfo_t structures. The implementor can choose to not implement this and return CADI_STATUS_CmdNotSupported if no instantiation time parameters exist.

Parameters

parameter_-info_list	This must point to an array of CADIParameterInfo_t structures with GetNumberOfParameters() elements. The implementor of the GetParameterInfos() method must fill in these structures. The id field must be set to a unique id to match up the parameter with the value passed back via the CADIParameterValue_t structures of the Instantiate() call. The isRunTime field should be set to false (=0). Depending on the type of the parameter either the minValue, maxValue and defaultValue fields (for integer parameters) or the defaultString field should be set.

Returns

Should return MTI_OK after filling in parameters. Can return CADI_STATUS_CmdNotSupported if the plugin does not have parameters.

5.8.2.5 virtual CAInterface∗ Instantiate (const char ∗ *instance_name,* uint32_t *number_of_parameters,* eslapi::CADIParameterValue_t ∗ *parameter_values*) `[pure virtual]`

This will instantiate a new instance of a trace plugin. Since it is possible to have multiple instances of the same trace plugin this method might be called multiple times, typically with different parameter values.

Parameters

instance_name	The name of the instance of the trace plugin. If a plugin is instantiated multiple times the names will be unique. This can for example be used to print better (error) messages, or to make file names unique based on the plugin instance name.
number_of_-parameters	This contains the number of parameter values passed. The number of parameter values passed can be between 0 and the total number of parameters as returned by GetNumberOfParameters().
parameter_-values	This contains the values configured for the instantiation time parameters. The mapping between parameter infos and parameter values is done via the parameter ids of these structures.

Returns

The ObtainInterface method of the CAInterface pointer returned should resolve into a PluginInstance class.

5.8.2.6 virtual void Release ( ) `[pure virtual]`

This method will be called to indicate the end of the lifetime of the PluginFactory. No further calls to any of its methods will be done.

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

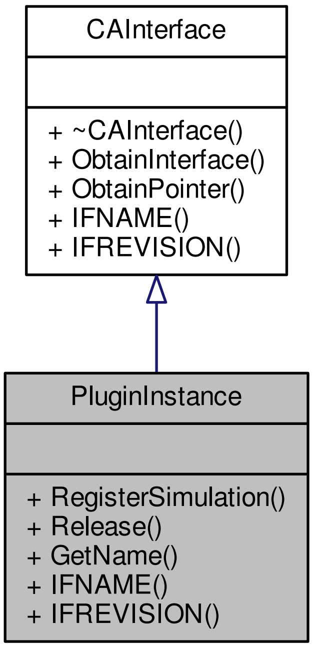
- PluginFactory_v0.h

5.9 PluginInstance Class Reference

Interface implementing the plugin interface.

`#include <PluginInstance_v0.h>`

Collaboration diagram for PluginInstance:



Public Member Functions

- virtual eslapi::CADIReturn_t RegisterSimulation (eslapi::CAInterface ∗simulation)=0
- virtual void Release ()=0
- virtual const char ∗ GetName () const =0

Static Public Member Functions

- static eslapi::if_name_t IFNAME ()
- static eslapi::if_rev_t IFREVISION ()

5.9.1 Detailed Description

Interface implementing the plugin interface.

This interface must be implemented by anybody implementing a (trace) plugin. The interface provides the RegisterSimulation() method which is called at instantiation time to pass a pointer to the main model trace interface of class SystemTraceInterface. An implementation of the class must implement the CAInterface::ObtainInterface() method, which will usually return a pointer to itself if an IFNAME() interface is requested.

5.9.2 Member Function Documentation

5.9.2.1 static eslapi::if_name_t IFNAME ( ) `[inline, static]`

Static helper method to return the interface name for the CAInterface.

Reimplemented from CAInterface.

5.9.2.2 static eslapi::if_rev_t IFREVISION ( ) `[inline, static]`

Static helper method to return the current interface revision for CAInterface.

Reimplemented from CAInterface.

5.9.2.3 virtual eslapi::CADIReturn_t RegisterSimulation (eslapi::CAInterface ∗ *simulation*) `[pure virtual]`

This method is called by the simulation at instantiation time.

Parameters

simulation	is a pointer of type CAInterface and can be turned into a SystemTraceInterface pointer by calling its ObtainInterface() method. It is also possible to get access to the CADI interface of the top level component by calling ObtainInterface() for a CADI interface on this simulation pointer.

Returns

This should usually return CADI_STATUS_OK to indicate that the plugin was successfully initialized.

5.9.2.4 virtual void Release ( ) `[pure virtual]`

After this method is called no further calls to a PluginInstance will be done. It should be used for any cleanup required, and can be used to delete the instance of PluginInstance, depending on the allocation model of the plugin.

5.9.2.5 virtual const char∗ GetName ( ) const `[pure virtual]`

Return the name if this plugin instance.

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

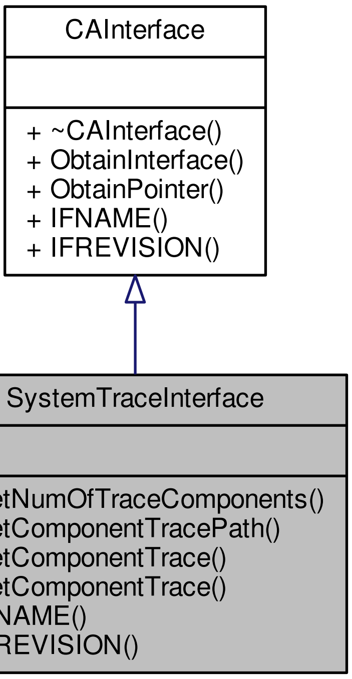
- PluginInstance_v0.h

5.10 SystemTraceInterface Class Reference

Class encapsulating all trace components of a system.

`#include <SystemTraceInterface_v1.h>`

Collaboration diagram for SystemTraceInterface:



Public Types

- typedef int TraceComponentIndex

Public Member Functions

- virtual TraceComponentIndex GetNumOfTraceComponents () const =0
- virtual const char ∗ GetComponentTracePath (TraceComponentIndex tci) const =0
- virtual eslapi::CAInterface ∗ GetComponentTrace (TraceComponentIndex tci) const =0
- virtual eslapi::CAInterface ∗ GetComponentTrace (const char ∗component_path) const =0

Static Public Member Functions

- static eslapi::if_name_t IFNAME ()
- static eslapi::if_rev_t IFREVISION ()

5.10.1 Detailed Description

Class encapsulating all trace components of a system.

This class is used to query which of a system's components provide trace sources. Components can be enumerated using the GetNumOfTraceComponents(), GetComponentTracePath(TraceComponentIndex) and GetComponentTrace(TraceComponentIndex) methods, or a component with known name can be requested using GetComponentTrace(const char ∗).

5.10.2 Member Typedef Documentation

5.10.2.1 typedef int TraceComponentIndex

This type is used to index the components which provide a ComponentTraceInterface.

5.10.3 Member Function Documentation

5.10.3.1 static eslapi::if_name_t IFNAME ( ) `[inline, static]`

Static helper method to return the interface name for the CAInterface.

Reimplemented from CAInterface.

5.10.3.2 static eslapi::if_rev_t IFREVISION ( ) `[inline, static]`

Static helper method to return the current interface revision for CAInterface.

Reimplemented from CAInterface.

5.10.3.3 virtual TraceComponentIndex GetNumOfTraceComponents ( ) const `[pure virtual]`

Return the number of components providing trace. The components can be accessed with indices 0 to GetNumOfTraceComponents()-1.

5.10.3.4 virtual const char∗ GetComponentTracePath (TraceComponentIndex *tci*) const `[pure virtual]`

Return the path of a component providing trace.

Parameters

tci	This must be between 0 and GetNumOfTraceComponents()-1.

Returns

Returns a string containing the path of the component. 0 will be returned if an illegal value for tci is given. The string returned lives as long as the simulation.

5.10.3.5 virtual eslapi::CAInterface∗ GetComponentTrace (TraceComponentIndex *tci*) const `[pure virtual]`

Get pointer to the ComponentTraceInterface class of a component by index.

Parameters

tci	This must be between 0 and GetNumOfTraceComponents()-1.

Returns

Returns a pointer to a ComponentTraceInterface instance. If an illegal tci is given then a 0 pointer is returned. The pointer returned is a CAInterface pointer. The ObtainInterface method must be used to convert it into a pointer of type TraceComponentInterface. The object returned will live as long as the simulation it belongs to.

5.10.3.6 virtual eslapi::CAInterface∗ GetComponentTrace (const char ∗ *component_path*) const `[pure virtual]`

Get a pointer to the ComponentTraceInterface class of a component by name.

Parameters

component_-path	The should be the pathname of a component providing trace sources as returned by GetComponentTracePath().

Returns

Returns a pointer to a ComponentTraceInterface instance. Returns 0 if no component with that pathname exists, or if the component does not provide trace sources. The pointer returned is a CAInterface pointer. The ObtainInterface method must be used to convert it into a pointer of type TraceComponentInterface. The object returned will live as long as the simulation it belongs to.

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

- SystemTraceInterface_v1.h

5.11 TraceSource Class Reference

```
#include <ComponentTraceInterface_v1.h>
```

Public Member Functions

- virtual SourceIndex GetIndex () const =0
- virtual const char ∗ GetName () const =0
- virtual const char ∗ GetDescription () const =0
- virtual FieldIndex GetNumFields () const =0
- virtual const EventFieldType ∗ GetField (FieldIndex index) const =0
- virtual const EventFieldType ∗ GetField (const char ∗name) const =0
- virtual EventClass ∗ CreateEventClass (FieldMask field_mask) const =0
- virtual Status RegisterCounter (uint64_t ∗counter)=0
- virtual Status UnregisterCounter (uint64_t ∗counter)=0
- virtual Status RegisterCounter (uint32_t ∗counter)=0
- virtual Status UnregisterCounter (uint32_t ∗counter)=0

Protected Member Functions

- virtual ~TraceSource ()

5.11.1 Detailed Description

A trace source is a single fixed source of a stream of events during simulation. It is possible to register a counter to be incremented when a trace event occurs. For more complex actions an EventClass should be created by selecting the set of fields which are of interest. A callback can then be registered with this EventClass to be called when the event occurs.

5.11.2 Constructor & Destructor Documentation

5.11.2.1 virtual ~TraceSource ( ) `[inline, protected, virtual]`

An interface user must never delete an instance of type TraceSource as returned by ComponentTraceInterface::GetTraceSource(). TraceSource instances live as long as their simulation.

5.11.3 Member Function Documentation

5.11.3.1 virtual const EventFieldType* GetField (FieldIndex *index*) const `[pure virtual]`

Return a instance of EventFieldType by index.

Parameters

index	The index must be between 0 and GetNumFields()-1.

Returns

Returns a pointer to an instance of EventFieldType. If the index is out of range 0 is returned.

5.11.3.2 virtual const EventFieldType* GetField (const char * *name*) const `[pure virtual]`

Return an instance of EventFieldType by name.

Parameters

name	The name of an event field as returned by EventFieldType::GetName().

Returns

Returns a pointer to an instance of EventFieldType. If no field with that name exists 0 is returned.

5.11.3.3 virtual EventClass* CreateEventClass (FieldMask *field_mask*) const `[pure virtual]`

Create an EventClass by selecting a subset of event fields.

Parameters

field_mask	

Returns

5.11.3.4 virtual Status RegisterCounter (uint64_t * *counter*) [pure virtual]

This will register a counter with an event source. Every time an trace event is triggered the counter will be increased by one. The interface user must provide the storage for the counter. It must remain valid until the simulation is destroyed, or the counter is unregistered by calling UnregisterCounter(). The counter variable will not be initialized when registering. The model must be halted to register a counter.

Parameters

counter	A pointer to a 64 bit counter variable allocated by the interface user. It's up to the user to keep the counter's memory allocated until it is unregistered.

Returns

Returns MTI_OK if successful.

5.11.3.5 virtual Status UnregisterCounter (uint64_t * *counter*) [pure virtual]

This method will unregister a counter which has previously been registered using RegisterCounter(). The model must be halted to unregister a counter.

Parameters

counter	Pointer to a 64 bit counter variable which has been used with RegisterCounter() earlier.

Returns

Will return MTI_OK if successful.

5.11.3.6 virtual Status RegisterCounter (uint32_t * *counter*) [pure virtual]

This will register a 32 bit counter with an event source. Every time an trace event is triggered the counter will be increased by one. The interface user must provide the storage for the counter. It must remain valid until the simulation is destroyed, or the counter is unregistered by calling UnregisterCounter(). The counter variable will not be initialized when registering. The model must be halted to register a counter.

Parameters

counter	A pointer to a 32 bit counter variable allocated by the interface user. It's up to the user to keep the counter's memory allocated until it is unregistered.

Returns

Returns MTI_OK if successful.

5.11.3.7 virtual Status UnregisterCounter (uint32_t * *counter*) `[pure virtual]`

This method will unregister a counter which has previously been registered using RegisterCounter(). The model must be halted to unregister a counter.

Parameters

counter	Pointer to a 32 bit counter variable which has been used with RegisterCounter() earlier.

Returns

Will return MTI_OK if successful.

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

- ComponentTraceInterface_v1.h

Chapter 6

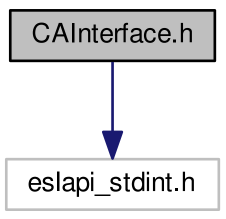
File Documentation

6.1 CAInterface.h File Reference

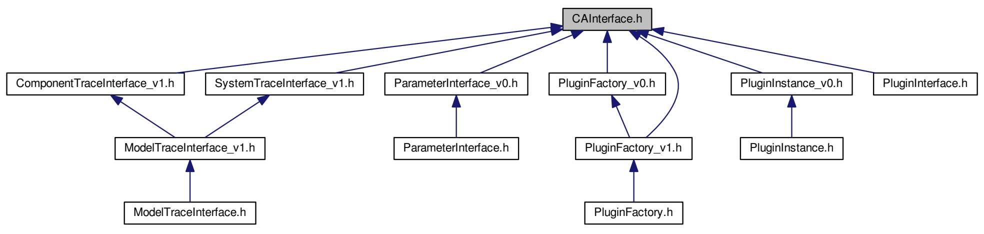
Definition of the abstract CAInterface class.

```
#include "eslapi_stdint.h"
```

Include dependency graph for CAInterface.h:



This graph shows which files directly or indirectly include this file:



Classes

- class CAInterface

Typedefs

- typedef char const ∗ if_name_t
- typedef uint32_t if_rev_t

6.1.1 Detailed Description

Definition of the abstract CAInterface class.

Date

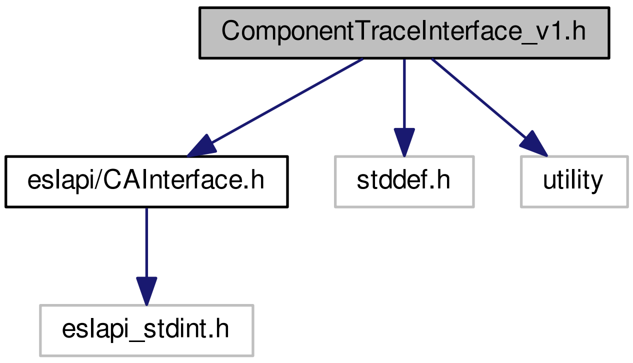
Copyright (c) 2006-2009 ARM. All Rights Reserved. Release 2.0.0

6.2 ComponentTraceInterface_v1.h File Reference

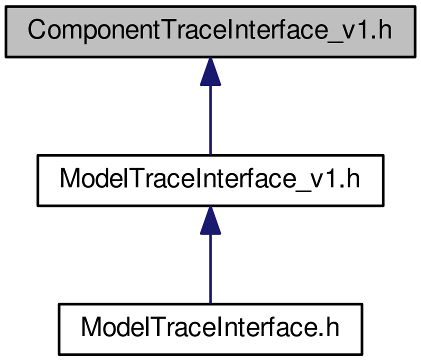
Version 1 MTI::ComponentTraceInterface and helper classes.

```
#include "eslapi/CAInterface.h"
#include <stddef.h>
#include <utility>
```

Include dependency graph for ComponentTraceInterface_v1.h:



This graph shows which files directly or indirectly include this file:



Classes

- class ComponentTraceInterface

  Class to access all the trace sources of a component.
- class EventFieldType
- class TraceSource
- class EventClass
- struct EventRecord

Typedefs

- typedef uint16_t EventClassId
- typedef int SourceIndex
- typedef int FieldIndex
- typedef int ValueIndex
- typedef uint32_t FieldMask
- typedef void(∗ CallbackT)(void ∗user_data, const EventClass ∗event_class, const EventRecord ∗event_record)
- typedef ComponentTraceInterface ModelTraceInterface

Enumerations

- enum **Status** {

  MTI_OK, MTI_FUNCTION_NOT_SUPPORTED, MTI_PARAMETER_NOT_SUPPORTED, MTI_-PARAMETER_INVALID,

  MTI_BUSY, MTI_ERROR }

6.2.1 Detailed Description

Version 1 MTI::ComponentTraceInterface and helper classes.

Date

Copyright ARM Limited 2008-2009 All Rights Reserved.

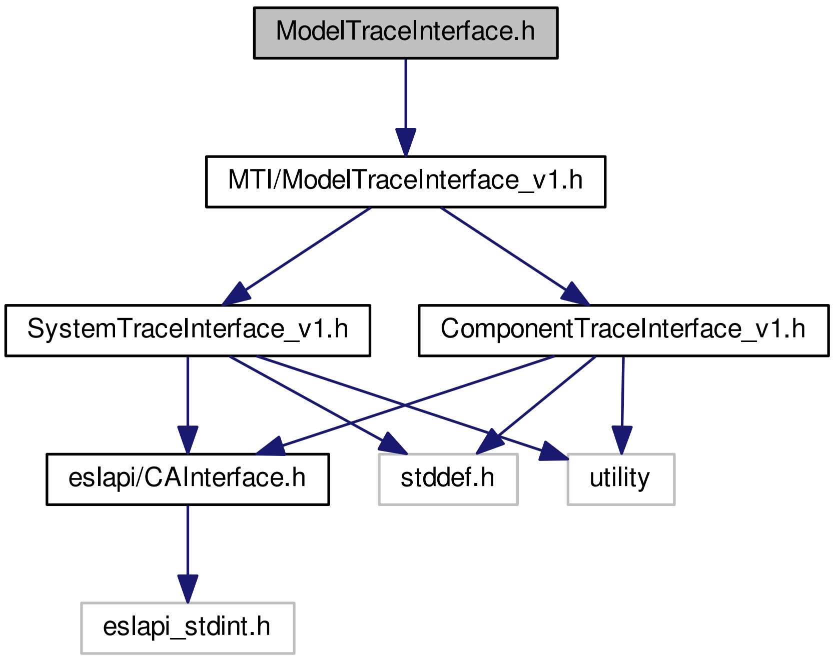
Normally this header file should not be included directly. The ModelTraceInterface.h header should be included instead.

6.3 ModelTraceInterface.h File Reference

The MTI::SystemTraceInterface and MTI::ComponentTraceInterface classes, and helper classes.

```
#include "MTI/ModelTraceInterface_v1.h"
```

Include dependency graph for ModelTraceInterface.h:


ModelTraceInterface.h
MTI/ModelTraceInterface_v1.h
SystemTraceInterface_v1.h
ComponentTraceInterface_v1.h
eslapi/CAInterface.h
stddef.h
utility
eslapi_stdint.h

6.3.1 Detailed Description

The MTI::SystemTraceInterface and MTI::ComponentTraceInterface classes, and helper classes.

Date

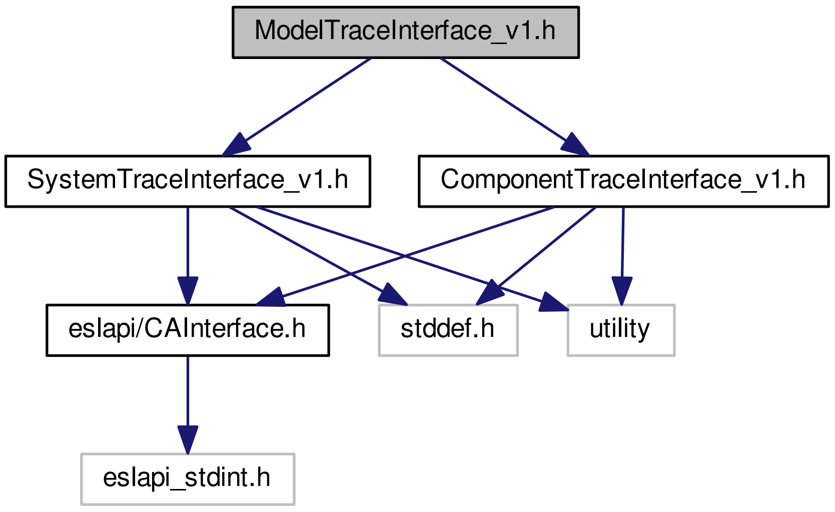
Copyright ARM Limited 2008-2009 All Rights Reserved.

6.4 ModelTraceInterface_v1.h File Reference

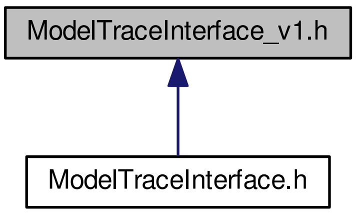
The version 1 of the SystemTraceInterface and ComponentTraceInterface classes and helper classes.

```
#include "SystemTraceInterface_v1.h"
#include "ComponentTraceInterface_v1.h"
```

Include dependency graph for ModelTraceInterface_v1.h:



This graph shows which files directly or indirectly include this file:



6.4.1 Detailed Description

The version 1 of the SystemTraceInterface and ComponentTraceInterface classes and helper classes.

Date

Copyright ARM Limited 2008-2009 All Rights Reserved.

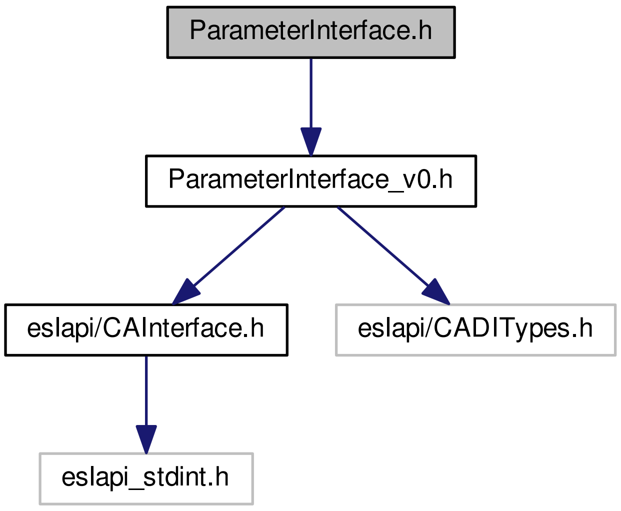
Normally this header file should not be included directly. The ModelTraceInterface.h header should be included instead.

6.5 ParameterInterface.h File Reference

Defining the runtime MTI::ParameterInterface.

`#include "ParameterInterface_v0.h"`

Include dependency graph for ParameterInterface.h:



6.5.1 Detailed Description

Defining the runtime MTI::ParameterInterface.

Date

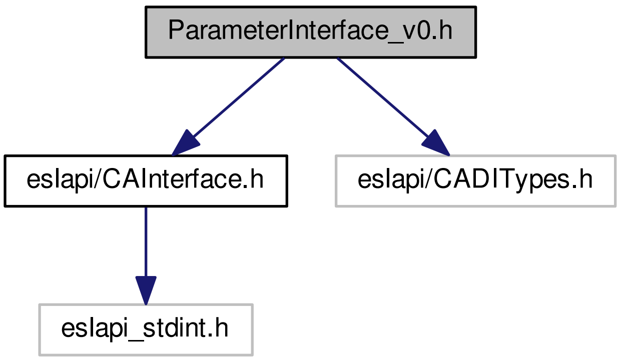
Copyright ARM Limited 2008-2009 All Rights Reserved.

6.6 ParameterInterface_v0.h File Reference

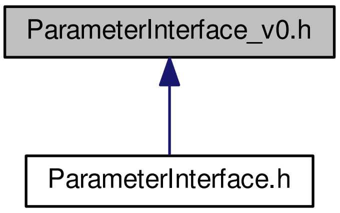
The runtime MTI::ParameterInterface, version 0.

```
#include "eslapi/CAInterface.h"
#include "eslapi/CADITypes.h"
```

Include dependency graph for ParameterInterface_v0.h:



This graph shows which files directly or indirectly include this file:



Classes

- class ParameterInterface

6.6.1 Detailed Description

The runtime MTI::ParameterInterface, version 0.

Date

Copyright ARM Limited 2008-2009 All Rights Reserved.

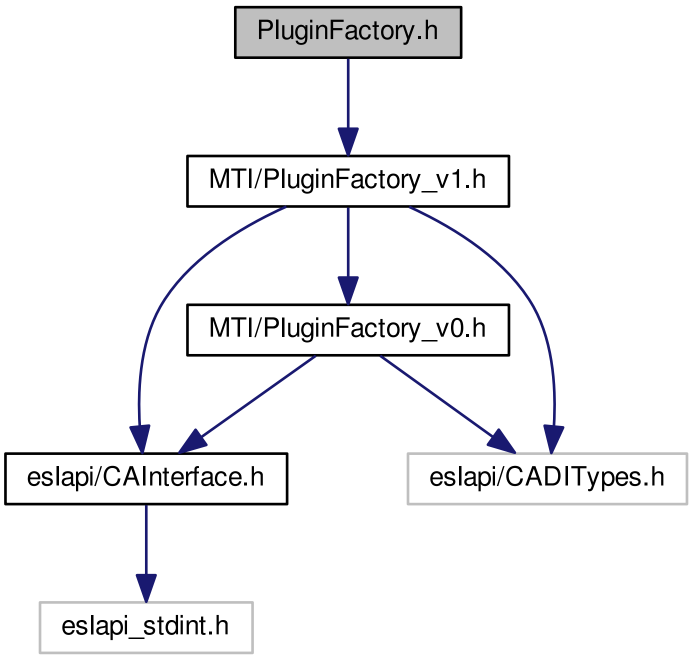
Normally this header file should not be included directly. The ParameterInterface.h header should be included instead.

6.7 PluginFactory.h File Reference

Declares the MTI::PluginFactory interface.

`#include "MTI/PluginFactory_v1.h"`

Include dependency graph for PluginFactory.h:



6.7.1 Detailed Description

Declares the MTI::PluginFactory interface.

Date

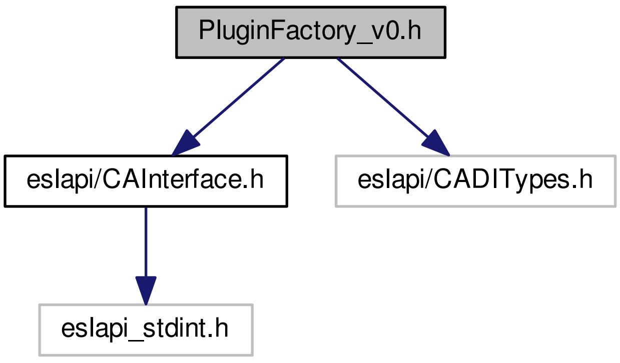
Copyright ARM Limited 2009 All Rights Reserved.

6.8 PluginFactory_v0.h File Reference

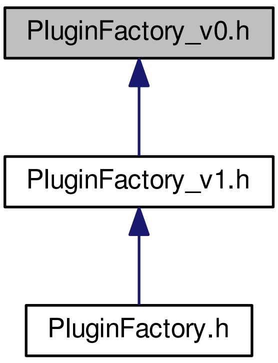
Declares the version 0 of the MTI::PluginFactory interface.

```
#include "eslapi/CAInterface.h"
#include "eslapi/CADITypes.h"
```

Include dependency graph for PluginFactory_v0.h:



This graph shows which files directly or indirectly include this file:



Classes

- class PluginFactory

  Class providing a method to instantiate a PluginInstance.

6.8.1 Detailed Description

Declares the version 0 of the MTI::PluginFactory interface.

Date

Copyright ARM Limited 2009 All Rights Reserved.

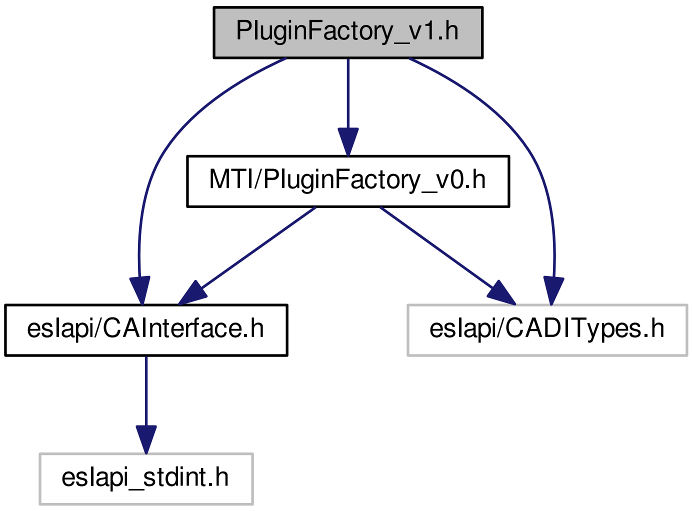
Normally this header file should not be included directly. The PluginFactory.h header should be included instead.

6.9 PluginFactory_v1.h File Reference

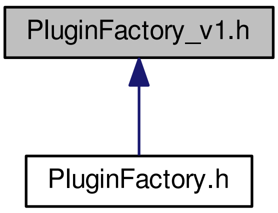
Declares the version 1 of the MTI::PluginFactory interface.

```
#include "eslapi/CAInterface.h"
#include "eslapi/CADITypes.h"
#include "MTI/PluginFactory_v0.h"
```

Include dependency graph for PluginFactory_v1.h:



This graph shows which files directly or indirectly include this file:



Classes

- class PluginFactory

6.9.1 Detailed Description

Declares the version 1 of the MTI::PluginFactory interface.

Date

Copyright ARM Limited 2011 All Rights Reserved.

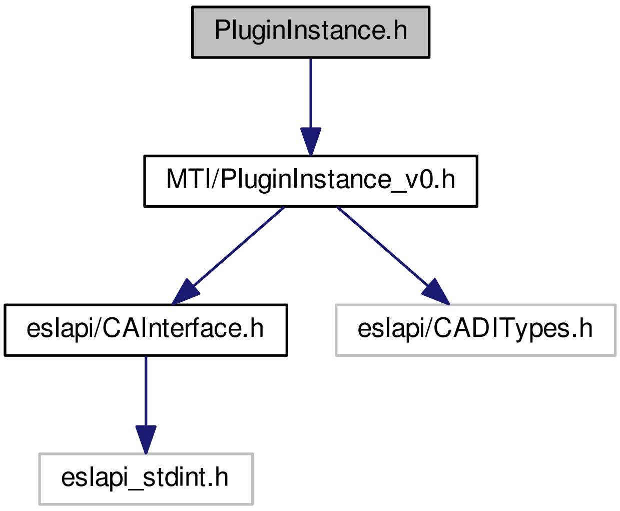
Normally this header file should not be included directly. The PluginFactory.h header should be included instead.

6.10 PluginInstance.h File Reference

Declares the MTI::PluginInstance interface.

```
#include "MTI/PluginInstance_v0.h"
```

Include dependency graph for PluginInstance.h:



6.10.1 Detailed Description

Declares the MTI::PluginInstance interface.

Date

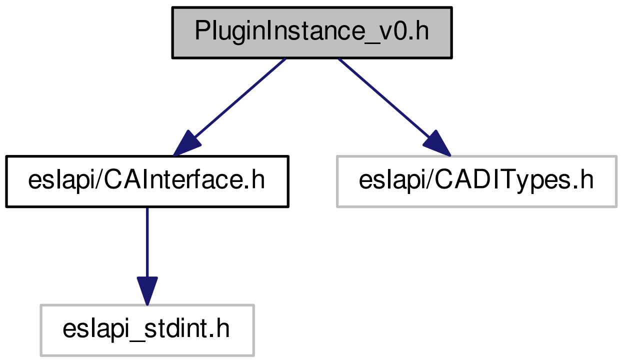
Copyright ARM Limited 2009 All Rights Reserved.

6.11 PluginInstance_v0.h File Reference

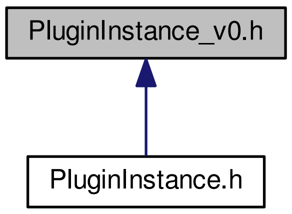
Declaring the version 0 of the MTI::PluginInstance interface.

```
#include "eslapi/CAInterface.h"
#include "eslapi/CADITypes.h"
```

Include dependency graph for PluginInstance_v0.h:



This graph shows which files directly or indirectly include this file:



Classes

- class PluginInstance

  Interface implementing the plugin interface.

6.11.1 Detailed Description

Declaring the version 0 of the MTI::PluginInstance interface.

Date

Copyright ARM Limited 2009 All Rights Reserved.

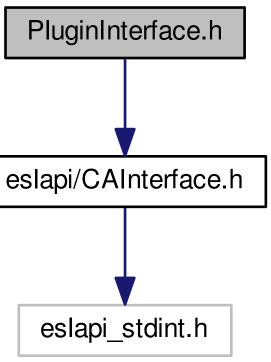
Normally this header file should not be included directly. The PluginInstance.h header should be included instead.

6.12 PluginInterface.h File Reference

Defines a generic shared library entry point.

`#include "eslapi/CAInterface.h"`

Include dependency graph for PluginInterface.h:



Typedefs

- typedef eslapi::CAInterface ∗(∗ GetCAInterfaceFunc)()

Functions

- PLUGIN_INTERFACE_WEXP eslapi::CAInterface ∗ GetCAInterface ()

6.12.1 Detailed Description

Defines a generic shared library entry point.

Date

Copyright ARM Limited 2008-2009 All Rights Reserved.

6.12.2 Typedef Documentation

6.12.2.1 typedef eslapi::CAInterface∗(∗ GetCAInterfaceFunc)()

Type describing the function prototype of the GetCAInterface function. Typical use is to cast to symbol pointer to a function pointer after dynamically loading a (trace) plugin.

6.12.3 Function Documentation

6.12.3.1 PLUGIN_INTERFACE_WEXP eslapi::CAInterface∗ GetCAInterface ( )

This is the entry point and only function of a trace plugin .so/.dll. It will return a CAInterface pointer, on which the ObtainInterface method must be called, to obtain an interface of type PluginFactory.

6.13 SystemTraceInterface_v1.h File Reference

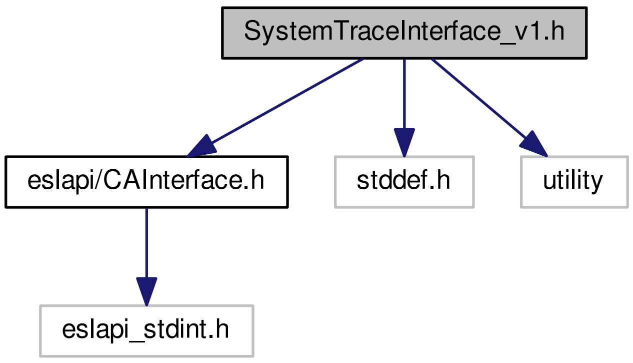
Version 1 SystemTraceInterface class.

`#include "eslapi/CAInterface.h"`

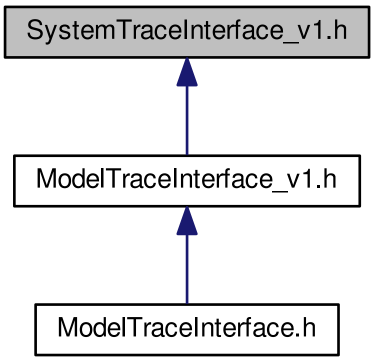
`#include <stddef.h>`

`#include <utility>`

Include dependency graph for SystemTraceInterface_v1.h:



This graph shows which files directly or indirectly include this file:



Classes

- class SystemTraceInterface

  Class encapsulating all trace components of a system.

6.13.1 Detailed Description

Version 1 SystemTraceInterface class.

Date

Copyright ARM Limited 2009 All Rights Reserved.

Normally this header file should not be included directly. The ModelTraceInterface.h header should be included instead.

Index