Saftlib

Types

The saftlib interfaces documentation uses a number of types which are character encoded. Because Saftlib used DBus in the past, the encoding is similar but not identical to the DBus type encoding. The following table lists all types and their representation in the C++ API:

y	uint8_t
b	bool
n	int16_t
q	uint16_t
i	int32_t
u	uint32_t
x	int64_t
t	uint64_t
d	double
T	saftlib::Time
s	std::string
aX	std::vector<X> where X is can be any type
a{KV}	std::map<K,V> K and V are are key and value types

de.gsi.saftlib.ActionSink

Name

de.gsi.saftlib.ActionSink — An output through which actions flow.

Methods

```
ToggleActive ();
ReadFill     (OUT q result);
```

Signals

```
Late        (u count,
             t event,
             t param,
             t deadline,
             t executed);
SigLate     (u count,
             t event,
             t param,
             T deadline,
             T executed);
Early       (u count,
             t event,
             t param,
             t deadline,
             t executed);
SigEarly    (u count,
             t event,
             t param,
             T deadline,
             T executed);
Conflict    (t count,
             t event,
             t param,
             t deadline,
             t executed);
SigConflict (t count,
             t event,
             t param,
             T deadline,
             T executed);
```

```
Delayed     (t count,
             t event,
             t param,
             t deadline,
             t executed);
SigDelayed  (t count,
             t event,
             t param,
             T deadline,
             T executed);
```

Properties

AllConditions	readable	as
ActiveConditions	readable	as
InactiveConditions	readable	as
MinOffset	readwrite	x
MaxOffset	readwrite	x
Latency	readable	t
EarlyThreshold	readable	t
Capacity	readable	q
MostFull	readwrite	q
SignalRate	readwrite	t
OverflowCount	readwrite	t
ActionCount	readwrite	t
LateCount	readwrite	t
EarlyCount	readwrite	t
ConflictCount	readwrite	t
DelayedCount	readwrite	t

Description

Conditions created on this ActionSink specify which timing events are translated into actions. These actions have execution timestamps which determine when the action is to be executed, precise to the nanosecond.

More specialized versions of this interface provide the 'NewCondition' method to create conditions specific to the type of ActionSink. For example, SoftwareActionSinks create conditions that emit signals to software. This interface captures the functionality common to all ActionSinks, such as atomic toggle, offset constraints, and introspection. In particular, ActionSinks have common failure modes.

Actions are queued for delivery at the appropriate time, in hardware. Hardware has limited storage, reflected by the Fill, Capacity, and MostFull properties. These should be monitored to ensure that the queue never overflows.

The first failure mode of an ActionSink is that the queue overflows. In this case, the hardware drops an action. Obviously, this is a critical error which may result in an undefined state. To prevent these failures, MostFull should be kept below some safety margin with respect to Capacity. Note: several distinct ActionSinks may share underlying hardware, and the Fill property is shared amongst all instances. Each overflow is recorded in the OverflowCount register. Due the rate at which this counter might increase, the API throttles updates using the SignalRate property.

Another critical error for an ActionSink is the possibility of a late action. This indicates that hardware was instructed to execute an action at a time in the past. This is typically caused by either a malfunctioning data master, desynchronized clocks, or conditions with large negative offsets. This is a critical failure as it might leave the system in an undefined state. Conditions may be configured to either drop or execute late actions. If late actions are dropped, a magnet might never be turned off. If late actions are executed, a magnet might be turned on again after it was supposed to be turned off (ie: the actions get misordered). In any case, LateCount is increased.

Similar to late actions, one can also have early actions. If an action is scheduled for execution too far into the future, the timing receiver will choose to mark it as early. This prevents these actions from permanently consuming space in the finite hardware buffers. Early actions are also critical failures as it can leave the system in an undefined state, just as a late action. Conditions may be configured to either drop or execute early actions.

The final misordering failure for an ActionSink is the possibilty of a Conflict. If two actions are scheduled to be executed at the same nanosecond, their relative order is undefined. These conflicts are a critical error as they may leave the system in an undefined state, just as with early and late actions. Conflicts should be prevented by never creating two Conditions on the same ActionSink which could occur at the same time. Note that it is NOT a Conflict for two actions to be executed at the same time by two different ActionSinks. For software, this means that two programs, each with their own SoftwareActionSink do not need to be concerned about conflicts between their schedules. As another example, two different LEMO output cables (corresponding to two OutputActionSinks) can be toggled high at the same time.

Finally, these is the possibility of a delayed action. Unlike late, early, and conflicting actions, delayed actions are never misordered. Thus, delayed actions are typically not as severe a failure mode, and Conditions default to allowing their execution. Delays can happen when the delivery rate of actions (potentially 1GHz) exceeds the capability of the receiver to process the actions. For example, an output might require 100ns to deliver an action. If two actions are scheduled for delivery back-to-back with 1ns between, the second action is delayed. For SoftwareActionSinks, delays can probably always be ignored because the handler is much much slower than the hardware. For a kicker, on the other hand, a delay is probably a critical error.

Method Details

The ToggleActive() method

```
ToggleActive ();
```

Atomically toggle the active status of conditions.

When reconfiguring an ActionSink, it is sometimes necessary to apply many changes simultaneously. To achieve this, simply use NewCondition to create all the new conditions in the inactive state. Then use this method to simultaneous toggle all conditions on this ActionSink. The new active conditions will be applied such that on one nanosecond, the old set is active and on the next nanosecond the new set is active.

The ReadFill() method

```
ReadFill (OUT q result);
```

Report the number of currently pending actions.

The timing receiver hardware can only queue a limited number of actions. This method reports the current number of queued actions, which includes actions from all users of the underlying hardware. So two programs each using a SoftwareActionSink will potentially see an increase in this value when both programs are active. Also, this value can change very rapidly and is thus changes are not signalled. Polling it is likely to miss short fluctuations. See MostFull for a better approach to monitoring.

'`OUT q result`':
: Number of pending actions.

Signal Details

The "Late" signal

```
Late (u count,
      t event,
      t param,
      t deadline,
      t executed);
```

Deprecated! Use SigLate instead.

Keep in mind that an action is only counted as late if it is scheduled for the past. An action which leaves the timing receiver after its deadline, due to a slow consumer, is a delayed (not late) action.

'`u count`': The new value of LateCount when this signal was raised.

‘t event’: The event identifier of the late action.

‘t param’: The parameter field, whose meaning depends on the event ID.

‘t deadline’:
The desired execution timestamp (event time + offset).

‘t executed’:
The actual execution timestamp.

The "SigLate" signal

```
SigLate (u count,
         t event,
         t param,
         T deadline,
         T executed);
```

‘u count’:

‘t event’:

‘t param’:

‘T deadline’:
‘T executed’:

The "Early" signal

```
Early (u count,
       t event,
       t param,
       t deadline,
       t executed);
```

Deprecated! Use SigEarly instead.

‘u count’: The new value of LateCount when this signal was raised.

‘t event’: The event identifier of the early action.

‘t param’: The parameter field, whose meaning depends on the event ID.

‘t deadline’:
The desired execution timestamp (event time + offset).

‘t executed’:
The actual execution timestamp.

The "SigEarly" signal

```
SigEarly (u count,
          t event,
          t param,
          T deadline,
          T executed);
```

'u count':

't event':

't param':

'T deadline':
'T executed':

The "Conflict" signal

```
Conflict (t count,
          t event,
          t param,
          t deadline,
          t executed);
```

Deprecated! use SigConflict instead.

't count': The new value of ConflictCount when this signal was raised.

't event': The event identifier of a conflicting action.

't param': The parameter field, whose meaning depends on the event ID.

't deadline':
The scheduled action execution timestamp (event time + offset).

't executed':
The timestamp when the action was actually executed.

The "SigConflict" signal

```
SigConflict (t count,
             t event,
             t param,
             T deadline,
             T executed);
```

't count':

‘t event’:

‘t param’:

‘T deadline’:
‘T executed’:

The "Delayed" signal

```
Delayed (t count,
         t event,
         t param,
         t deadline,
         t executed);
```

Deprecated! Use SigDelayed instead.

‘t count’: The value of DelayedCount when this signal was raised.

‘t event’: The event identifier of the delayed action.

‘t param’: The parameter field, whose meaning depends on the event ID.

‘t deadline’:
The desired execution timestamp (event time + offset).

‘t executed’:
The timestamp when the action was actually executed.

The "SigDelayed" signal

```
SigDelayed (t count,
            t event,
            t param,
            T deadline,
            T executed);
```

‘t count’:

‘t event’:

‘t param’:

‘T deadline’:
‘T executed’:

Property Details

The "AllConditions" property

```
AllConditions  readable  as
```

All conditions created on this ActionSink.

All active and inactive conditions on the ActionSink. Each object path implements the the matching Condition interface; for example, a SoftwareActionSink will have SoftwareConditions.

The "ActiveConditions" property

```
ActiveConditions  readable  as
```

All active conditions created on this ActionSink.

All active conditions on the ActionSink. Each object path implements the the matching Condition interface; for example, a SoftwareActionSink will have SoftwareConditions.

The "InactiveConditions" property

```
InactiveConditions  readable  as
```

All inactive conditions created on this ActionSink.

All inactive conditions on the ActionSink. Each object path implements the the matching Condition interface; for example, a SoftwareActionSink will have SoftwareConditions.

The "MinOffset" property

```
MinOffset  readwrite  x
```

Minimum allowed offset (nanoseconds) usable in NewCondition.

Large offsets are almost always an error. A very negative offset could result in Late actions. By default, no condition may be created with an offset smaller than -100us. Attempts to create conditions with offsets less than MinOffset result in an error. Change this property to override this safety feature.

The "MaxOffset" property

```
MaxOffset  readwrite  x
```

Maximum allowed offset (nanoseconds) usable in NewCondition.

Large offsets are almost always an error. A large positive offset could result in early actions being created. By default, no condition may have an offset larger than 1s. Attempts to create conditions with offsets greater than MaxOffset result in an error. Change this property to override this safety feature.

The "Latency" property

```
Latency  readable   t
```

Nanoseconds between event and earliest execution of an action.

The "EarlyThreshold" property

```
EarlyThreshold  readable   t
```

Actions further into the future than this are early.

If an action is scheduled for execution too far into the future, it gets truncated to at most EarlyThreshold nanoseconds into the future.

The "Capacity" property

```
Capacity  readable   q
```

The maximum number of actions queueable without Overflow.

The timing receiver hardware can only queue a limited number of actions. This property reports the maximum number of actions that may be simultaneously queued. Be aware that this resource may be shared between multiple ActionSinks. For example, all SoftwareActionSinks share a common underlying queue in hardware. This Capacity represents the total size, which must be shared.

The "MostFull" property

```
MostFull  readwrite  q
```

Report the largest number of pending actions seen.

The timing receiver hardware can only queue a limited number of actions. This property reports the highest Fill level seen by the hardware since it was last reset to 0. Keep in mind that the queue may be shared, including actions from all users of the underlying hardware. So two programs each using a SoftwareActionSink will potentially see an increase in this value when both programs are active.

The "SignalRate" property

```
SignalRate  readwrite  t
```

Minimum interval between updates (nanoseconds, default 100ms).

The properties OverflowCount, ActionCount, LateCount, EarlyCount, ConflictCount, and DelayedCount can increase rapidly. To prevent excessive CPU load, SignalRate imposes a minimum cooldown between updates to these values.

The "OverflowCount" property

```
OverflowCount  readwrite  t
```

The number of actions lost due to Overflow.

The underlying hardware queue may overflow once Fill=Capacity. This is a critical error condition that must be handlded. The causes may be either: 1- the actions have an execution time far enough in the future that too many actions are buffered before they are executed, or 2- the receiving component is unable to execute actions as quickly as the timing system delivers them. The second case is particularly likely for SoftwareActionSinks that attempt to listen to high frequency events. Even though SoftwareActionSinks share a common queue, Overflow is reported only to the ActionSink whose action was dropped.

As overflows can occur very rapidly, OverflowCount may increase by more than 1 between emissions. There is a minimum delay of SignalRate nanoseconds between updates to this property.

The "ActionCount" property

```
ActionCount  readwrite  t
```

The number of actions processed by the Sink.

As actions can be emitted very rapidly, ActionCount may increase by more than 1 between emissions. There is a minimum delay of SignalRate nanoseconds between updates to this property.

The "LateCount" property

```
LateCount  readwrite  t
```

The number of actions delivered late.

As described in the interface overview, an action can be late due to a buggy data master, loss of clock synchronization, or very negative condition offsets. This is a critical failure as it can result in misordering of executed actions. Each such failure increases this counter.

As late acitons can occur very rapidly, LateCount may increase by more than 1 between emissions. There is a minimum delay of SignalRate nanoseconds between updates to this property.

The "EarlyCount" property

```
EarlyCount  readwrite  t
```

The number of actions delivered early.

As described in the interface overview, an action can be early due to a buggy data master, loss of clock synchronization, or very positive condition offsets. This is a critical failure as it can result in misordering of executed actions. Each such failure increases this counter.

As early acitons can occur very rapidly, EarlyCount may increase by more than 1 between emissions. There is a minimum delay of SignalRate nanoseconds between updates to this property.

The "ConflictCount" property

```
ConflictCount  readwrite  t
```

The number of actions which conflicted.

If two actions should have been executed simultaneously by the same ActionSink, they are executed in an undefined order. Each time this happens, the ConflictCount is increased.

As conflicts can occur very rapidly, ConflictCount may increase by more than 1 between emissions. There is a minimum delay of SignalRate nanoseconds between updates to this property.

The "DelayedCount" property

```
DelayedCount  readwrite  t
```

The number of actions which have been delayed.

The timing receiver emits actions potentially every nanosecond. In the case that the receiver cannot immediately process an action, the timing receiver delays the action until the receiver is ready. This can happen either because the receiver was still busy with a previous action or the output was used externally (bus arbitration).

As actions can be emitted very rapidly, DelayedCount may increase by more than 1 between emissions. There is a minimum delay of SignalRate nanoseconds between updates to this property.

de.gsi.saftlib.Condition

Name

de.gsi.saftlib.Condition — A rule matched against incoming events

Properties

```
ID              readwrite  t
Mask            readwrite  t
Offset          readwrite  x
AcceptLate      readwrite  b
AcceptEarly     readwrite  b
AcceptConflict  readwrite  b
AcceptDelayed   readwrite  b
Active          readwrite  b
```

Description

Conditions are created for ActionSinks to select which events the sink should respond to. Different ActionSinks return handles to different Conditions. This interface is the common denominator.

Property Details

The "ID" property

```
ID  readwrite  t
```

The event identifier which this condition matches against.

An incoming event matches if its ID and this ID agree on all the bits set in this condition's Mask.

The "Mask" property

```
Mask  readwrite  t
```

The mask used when comparing event IDs.

An incoming event matches if its ID and the ID property of this Condition agree on all the bits set in this Mask.

The "Offset" property

```
Offset  readwrite  x
```

Added to an event's time to calculate the action's time.

The "AcceptLate" property

```
AcceptLate  readwrite  b
```

Should late actions be executed? Defaults to false

The "AcceptEarly" property

```
AcceptEarly  readwrite  b
```

Should early actions be executed? Defaults to false

The "AcceptConflict" property

```
AcceptConflict  readwrite  b
```

Should conflicting actions be executed? Defaults to false

The "AcceptDelayed" property

```
AcceptDelayed  readwrite  b
```

Should delayed actions be executed? Defaults to true

The "Active" property

```
Active  readwrite  b
```

The condition should be actively matched against events.

An inactive condition is not used to match against events. You can toggle the active state of this condition via this property, or if multiple Conditions should be atomically adjusted, use the ToggleActive method on the ActionSink.

de.gsi.saftlib.Device

Name

de.gsi.saftlib.Device — A device attached to saftd via Etherbone.

Methods

```
Remove ();
```

Properties

```
EtherbonePath  readable   s
Name           readable   s
```

Description

Devices are attached to saftlib by specifying a name and a path. The name should denote the logical relationship of the device to saftd. For example, baseboard would be a good name for the timing receiver attached to an SCU. If an exploder is being used to output events to an oscilloscope, a good logical name might be scope. In these examples, the path for the SCU baseboard would be dev/wbm0, and the scope exploder would be dev/ttyUSB3 or similar.

This scheme is intended to make it easy to hot-swap hardware. If the exploder dies, you can simply attach a new one under the same logical name, even though the path might be different.

Method Details

The Remove() method

```
Remove ();
```

Remove the device from saftlib management.

Property Details

The "EtherbonePath" property

```
EtherbonePath  readable   s
```

The path through which the device is reached.

The "Name" property

```
Name  readable   s
```

The logical name with which the device was connected.

de.gsi.saftlib.EmbeddedCPUActionSink

Name

de.gsi.saftlib.EmbeddedCPUActionSink — An output through which EmbeddedCPU actions flow.

Methods

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  u tag,
              OUT s result);
```

Description

This inteface allows the generation of SCU timing events. A EmbeddedCPUActionSink is also an ActionSink and Owned.

If two SoftwareConditions are created on the same SoftwareActionSink which require simultaneous delivery of two Actions, then they will be delivered in arbitrary order, both having the 'conflict' flag set.

Method Details

The NewCondition() method

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  u tag,
              OUT s result);
```

Create a condition to match incoming events

This method creates a new condition that matches events whose identifier lies in the range [id & mask, id | ~mask]. The offset acts as a delay which is added to the event's

execution timestamp to determine the timestamp when the matching condition fires its action. The returned object path is a EmbeddedCPUCondition object.

'`IN b active`':
: Should the condition be immediately active

'`IN t id`': Event ID to match incoming event IDs against

'`IN t mask`':
: Set of bits for which the event ID and id must agree

'`IN x offset`':
: Delay in nanoseconds between event and action

'`IN u tag`':
: The 32-bit value to send on the EmbeddedCPU

'`OUT s result`':
: Object path to the created EmbeddedCPUCondition

de.gsi.saftlib.EmbeddedCPUCondition

Name

de.gsi.saftlib.EmbeddedCPUCondition — Matched against incoming events on a EmbeddedCPUActionSink.

Properties

```
Tag  readwrite  u
```

Description

EmbeddedCPUConditions are created by EmbeddedCPUActionSinks to select which events should generate callbacks. This interface always implies that the object also implements the general Condition interface.

Property Details

The "Tag" property

```
Tag  readwrite  u
```

The tag which is sent to the EmbeddedCPU by this condition

de.gsi.saftlib.EventSource

Name

de.gsi.saftlib.EventSource — An external source of timing events.

Properties

```
Resolution   readable   t
EventBits    readable   u
EventEnable  readwrite  b
EventPrefix  readwrite  t
```

Description

Normally, timing events come from the data master in the timing network. However, external inputs to the timing receiver can also be used as a source of timing events. For example, a LEMO input can be configured to generate a timing event on edge transitions.

When the external trigger occurs, it is timestamped and an event is created which will be processed as normal by the timing receiver. The event identifier comes from the EventPrefix. However, some external devices additionally provide data which is filled into the low EventBits of the generated event identifier.

Property Details

The "Resolution" property

```
Resolution  readable   t
```

The precision of generated timestamps in nanoseconds.

The "EventBits" property

```
EventBits  readable   u
```

How many bits of external data are included in the ID

The "EventEnable" property

```
EventEnable  readwrite  b
```

Should the event source generate events

The "EventPrefix" property

```
EventPrefix  readwrite  t
```

Combined with low EventBits to create generated IDs

de.gsi.saftlib.FunctionGenerator

Name

de.gsi.saftlib.FunctionGenerator — Function Generator for creating timing triggered waveforms

Methods

```
Arm                         ();
Abort                       ();
ReadFillLevel               (OUT t  result);
AppendParameterSet          (IN  an coeff_a,
                             IN  an coeff_b,
                             IN  ai coeff_c,
                             IN  ay step,
                             IN  ay freq,
                             IN  ay shift_a,
                             IN  ay shift_b,
                             OUT b  low_fill);
Flush                       ();
ReadExecutedParameterCount (OUT u  result);
```

Signals

```
SigEnabled (b enabled);
SigArmed   (b armed);
SigRunning (b running);
Started    (t time);
SigStarted (T time);
Stopped    (t time,
            b aborted,
            b hardwareMacroUnderflow,
            b microControllerUnderflow);
SigStopped (T time,
            b aborted,
            b hardwareMacroUnderflow,
            b microControllerUnderflow);
Refill     ();
```

Properties

Version	readable	u
SCUbusSlot	readable	y
DeviceNumber	readable	y
OutputWindowSize	readable	y
Enabled	readable	b
Armed	readable	b
Running	readable	b
StartTag	readwrite	u

Description

The function generator creates a waveform from a piecewise continuous sequence of second order polynomials. These polynomials are specified using a fixed-width "parameter tuple" format. Once triggered, the function generator outputs samples created from each tupled polynomial until it exhausts the specified piecewise width of the polynomial, whereupon it begins outputting samples from the next tupled polynomial.

The general work-flow to use the FunctionGenerator is to append sets of tuples describing the waveform using the AppendParameterSet function. Then, select a SCUbus timing tag (StartTag) whose appearance should trigger waveform generation. Finally, the FunctionGenerator is armed by calling Arm, whereafter it may be triggered manually via the SCUbusActionSink->InjectTag method, or by a timing event sent by the data master, which is configured to create the StartTag.

If you need an unending waveform, follow the steps above, but additionally monitor the Refill signal, and append additional parameter tuples until low_fill is false.

The sequence of signals of successful waveform generation occur in this order: Enabled(true) Armed(true) Armed(false) Started Running(true) Refill Running(false) Stopped Enabled(false)

Method Details

The Arm() method

```
Arm ();
```

Enable the function generator and arm it.

A function generator can only be Armed if FillLevel is non-zero. An Enabled function generator can not be Armed again until it either completes waveform generation or the user

calls Abort. Arming a function generator takes time. Wait for Armed to transition to true before sending StartTag.

The Abort() method

```
Abort ();
```

Abort waveform generation.

This directs the hardware to stop waveform generation. If the function generator was Armed, it is disarmed and disabled, without outputting any waveform data. If the function generator is running, output is Stopped at the current value and disabled. Aborting a function generator takes time, so even after a call to Abort, the function generator might still be Started. However, it will reach the disabled state as quickly as it can, transitioning through Stopped as usual. If the Owner of a FunctionGenerator quits without running Disown, the Abort is run automatically.

The ReadFillLevel() method

```
ReadFillLevel (OUT t result);
```

Remaining waveform data in nanoseconds.

The SAFTd has sufficient parameters buffered to supply the function generator with data for the specified time in nanoseconds. Note, due to the slow nature of software, if the function generator is currently running, the read value will already be out-of-date upon return. This property should be used for informational use only.

'`OUT t result`':
: Remaining waveform data in nanoseconds.

The AppendParameterSet() method

```
AppendParameterSet (IN  an coeff_a,
                    IN  an coeff_b,
                    IN  ai coeff_c,
                    IN  ay step,
                    IN  ay freq,
                    IN  ay shift_a,
                    IN  ay shift_b,
                    OUT b  low_fill);
```

Append parameter tuples describing waveform to generate.

This function appends the parameter vectors (which must be equal in length) to the FIFO of remaining waveform to generate. Each parameter set (coefficients) describes a

number of output samples in the generated wave form. Parameter sets are executed in order until no more remain.

If the fill level is not high enough, this method returns true. Only once this function has returned false can you await the Refill signal.

At each step, the function generator outputs high_bits(c*2^32 + b*t + c*t*t), where t ranges from 0 to numSteps-1. high_bits are the high OutputWindowSize bits of the resulting 64-bit signed value.

'`IN an coeff_a`':
: Quadratic coefficient (a), 16-bit signed

'`IN an coeff_b`':
: Linear coefficient (b), 16-bit signed

'`IN ai coeff_c`':
: Constant coefficient (c), 32-bit signed

'`IN ay step`':
: Number of interpolated samples (0=250, 1=500, 2=1000, 3=2000, 4=4000, 5=8000, 6=16000, or 7=32000)

'`IN ay freq`':
: Output sample frequency (0=16kHz, 1=32kHz, 2=64kHz, 3=125kHz, 4=250kHz, 5=500kHz, 6=1MHz, or 7=2MHz)

'`IN ay shift_a`':
: Exponent of coeff_a, 6-bit unsigned; a = coeff_a*2^shift_a

'`IN ay shift_b`':
: Exponent of coeff_b, 6-bit unsigned; b = coeff_b*2^shift_b

'`OUT b low_fill`':
: Fill level remains too low

The Flush() method

```
Flush ();
```

Empty the parameter tuple set.

Flush may only be called when not Enabled. Flush does not clear the ExecutedParameterCount.

The ReadExecutedParameterCount() method

```
ReadExecutedParameterCount (OUT u result);
```

Number of parameter tuples executed by hardware.

This counts the total number of parameter tuples executed since the last Started signal. Obviously, if the function generator is running, the returned value will be old.

‘`OUT u result`’:
: Number tuples executed by hardware.

Signal Details

The "SigEnabled" signal

```
SigEnabled (b enabled);
```

‘`b enabled`’:

The "SigArmed" signal

```
SigArmed (b armed);
```

‘`b armed`’:

The "SigRunning" signal

```
SigRunning (b running);
```

‘`b running`’:

The "Started" signal

```
Started (t time);
```

Deprecated! use SigStarted instead.

This signal notifies software when function generation has begun, possibly to update a GUI or other user-facing status information.

‘`t time`’: Time when function generation began in nanoseconds since 1970

The "SigStarted" signal

```
SigStarted (T time);
```

Function generation has begun.

This signal notifies software when function generation has begun, possibly to update a GUI or other user-facing status information.

'`T time`': Time when function generation began in nanoseconds since 1970

The "Stopped" signal

```
Stopped (t time,
         b aborted,
         b hardwareMacroUnderflow,
         b microControllerUnderflow);
```

Deprecated! Use SigStopped instead.

'`t time`': Time when function generation ended in nanoseconds since 1970

'`b aborted`':
'`b hardwareMacroUnderflow`':
A fatal error, indicating the SCUbus is congested

'`b microControllerUnderflow`':
A fatal error, indicating the host CPU is overloaded

The "SigStopped" signal

```
SigStopped (T time,
            b aborted,
            b hardwareMacroUnderflow,
            b microControllerUnderflow);
```

Function generation has ended.

The function generator stops either successfully (when all data has been sent), or it stops due to an error. When an error occurs, the function generator stops and holds its most recent value. This can occur due to two causes:

hardwareMacroUnderflow, a fatal error indicating the hardware ran out of data. If the SCUbus is too busy, it can happen that the waveform data stored in the function generator HDL is not refilled in time. This error can only be mitigated by ensuring that the function generator does not share the SCUbus with other users.

microControllerUnderflow, a fatal error indicating the microcontroller ran out of data. If the host CPU running this software is too busy, it can happen that the waveform data is

not delivered to the microcontroller before the microcontroller runs out of data. This error can be mitigated by reducing the number of busy processes running on the system.

Once the function generator has stopped, ExecutedParameterCount remains valid until the next time the function generator starts. After stopping, regardless of if the generation was successful or not, the parameter FIFO is cleared, Enabled is false, and this signal emitted.

'`T time`': Time when function generation ended in nanoseconds since 1970

'`b aborted`':
'`b hardwareMacroUnderflow`':
A fatal error, indicating the SCUbus is congested

'`b microControllerUnderflow`':
A fatal error, indicating the host CPU is overloaded

The "Refill" signal

```
Refill ();
```

More tuples must be appended to ensure uninterrupted waveform.

In order to guarantee an uninterrupted supply of data to the function generator, there should be data buffered in the SAFTd. When the buffer fill level gets too low, this signal is emitted, and you should run AppendParameterSet. If you do not, function generation will cease, signalling successful completion of the waveform with Stopped.

Property Details

The "Version" property

```
Version  readable   u
```

Version of the hardware macro

The "SCUbusSlot" property

```
SCUbusSlot  readable   y
```

Slot number of the slave card

The "DeviceNumber" property

```
DeviceNumber  readable   y
```

number of the hardware macro inside of the slave card

The "OutputWindowSize" property

```
OutputWindowSize  readable   y
```

Number of bits output by the function generator

The "Enabled" property

```
Enabled  readable   b
```

Hardware is allowed to generate an output waveform.

If a function generator is disabled, its output is constant. Enabled is set to true using Arm and transitions to false either upon completion or after a call to Abort.

The "Armed" property

```
Armed  readable   b
```

Upon receipt of StartTag, output will begin.

If a function generator is Armed, it is also Enabled. Once waveform data has been loaded into the function generator, Arm it. Shortly thereafter (once the hardware is ready), the Armed property will transition to true, indicating the hardware is ready to react. Once StartTag is received, Armed changes to false and Started is emitted.

The "Running" property

```
Running  readable   b
```

The function generator is currently producing a waveform.

Function generation is started by sending a SCUbus tag which matches the StartTag property of an Armed function generator. When this property transitions to true, the Started signal is emitted. When this property transitions to false, the Stopped signal is emitted.

The "StartTag" property

```
StartTag  readwrite  u
```

The SCUbus tag which causes function generation to begin.

If the function generator is Armed and this tag is sent to the SCUbus, then the function generator will begin generating the output waveform. StartTag may only be set when the FunctionGenerator is not Enabled.

de.gsi.saftlib.FunctionGeneratorFirmware

Name

de.gsi.saftlib.FunctionGeneratorFirmware — Representation of the FunctionGenerator Firmware

Methods

```
Scan (OUT a{ss} fgs);
```

Properties

```
Version  readable   u
```

Description

long description

Method Details

The Scan() method

```
Scan (OUT a{ss} fgs);
```

Scan bus for fg channels.

‘`OUT a{ss} fgs`’:

Property Details

The "Version" property

```
Version  readable   u
```

Version of the hardware macro

de.gsi.saftlib.InputEventSource

Name

de.gsi.saftlib.InputEventSource — An input used to generate timing events

Methods

```
ReadInput (OUT b result);
```

Properties

StableTime	readwrite	u
InputTermination	readwrite	b
SpecialPurposeIn	readwrite	b
GateIn	readwrite	b
InputTerminationAvailable	readable	b
SpecialPurposeInAvailable	readable	b
LogicLevelIn	readable	s
IndexIn	readable	u
TypeIn	readable	s
Output	readable	s

Description

An InputEventSource is also an EventSource and Owned.

InputEventSources generate two events: high and low transition. These are encoded in the resulting event ID's lowest bit.

Method Details

The ReadInput() method

```
ReadInput (OUT b result);
```

Read the current input value.

For inoutputs, this may differ from the Output value, if OutputEnable is false. To receive a signal on Input changes, use the EventSource interface to create timing events and monitor these via a SoftwareActionSink.

'`OUT b result`':
: The current logic level on the input.

Property Details

The "StableTime" property

```
StableTime  readwrite  u
```

Deglitch threshold for the input

The number of nanoseconds a signal must remain high or low in order to be considered a valid transition. Increasing this value will not impact the resulting timestamps, but will hide transitions smaller than the threshold. For example, if StableTime=400, then a 5MHz signal would be completely ignored.

The "InputTermination" property

```
InputTermination  readwrite  b
```

Set the input termination

Some inputs need termination to receive a clean input signal. However, if the same IO is used as an Output, termination should probably be disabled. This defaults to on. See also OutputEnable if this is an inoutput.

The "SpecialPurposeIn" property

```
SpecialPurposeIn  readwrite  b
```

Is the special function enabled.

The "GateIn" property

```
GateIn  readwrite  b
```

Set input gate or get gate status.

The "InputTerminationAvailable" property

```
InputTerminationAvailable  readable  b
```

Can (input) termination be configured.

The "SpecialPurposeInAvailable" property

```
SpecialPurposeInAvailable  readable  b
```

Can special configuration be configured.

The "LogicLevelIn" property

```
LogicLevelIn  readable  s
```

Logic level of the input (LVDS, LVTTL, ...)

The "IndexIn" property

```
IndexIn  readable  u
```

IO index.

The "TypeIn" property

```
TypeIn  readable  s
```

IO type (GPIO, LVDS, ...)

The "Output" property

`Output readable s`

If non-empty, path of the Output object for the same physical IO

de.gsi.saftlib.MILbusActionSink

Name

de.gsi.saftlib.MILbusActionSink — An output through which MILbus actions flow.

Methods

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  q tag,
              OUT s result);
InjectTag    (IN  q tag);
```

Description

This inteface allows the generation of MIL timing events. A MILbusActionSink is also an ActionSink and Owned.

If two SoftwareConditions are created on the same SoftwareActionSink which require simultaneous delivery of two Actions, then they will be delivered in arbitrary order, both having the 'conflict' flag set.

Method Details

The NewCondition() method

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  q tag,
              OUT s result);
```

Create a condition to match incoming events

This method creates a new condition that matches events whose identifier lies in the range [id & mask, id | ~mask]. The offset acts as a delay which is added to the event's

execution timestamp to determine the timestamp when the matching condition fires its action. The returned object path is a MILBUSCondition object.

'`IN b active`':
: Should the condition be immediately active

'`IN t id`': Event ID to match incoming event IDs against

'`IN t mask`':
: Set of bits for which the event ID and id must agree

'`IN x offset`':
: Delay in nanoseconds between event and action

'`IN q tag`':
: The 16-bit value to send on the MILbus

'`OUT s result`':
: Object path to the created MILbusCondition

The InjectTag() method

```
InjectTag (IN  q tag);
```

Directly generate a MILbus timing event.

For debugging, it can be helpful to simply create MILbus events without a matching timing event.

'`IN q tag`':
: The 16-bit value to push to the MILbus.

de.gsi.saftlib.MILbusCondition

Name

de.gsi.saftlib.MILbusCondition — Matched against incoming events on a MILbusActionSink.

Properties

```
Tag  readwrite  q
```

Description

MILbusConditions are created by MILbusActionSinks to select which events should generate callbacks. This interface always implies that the object also implements the general Condition interface.

Property Details

The "Tag" property

```
Tag  readwrite  q
```

The tag which is sent to the MILbus by this condition

de.gsi.saftlib.MasterFunctionGenerator

Name

de.gsi.saftlib.MasterFunctionGenerator — Interface to multiple Function Generators

Methods

```
Arm                                 ();
Abort                               (IN  b   wait_for_abort_ack);
InitializeSharedMemory              (IN  s   shared_memory_name);
AppendParameterTuplesForBeamProcess (IN  i   beam_process,
                                     IN  b   arm,
                                     IN  b   wait_for_arm_ack);
AppendParameterSets                 (IN  aan coeff_a,
                                     IN  aan coeff_b,
                                     IN  aai coeff_c,
                                     IN  aay step,
                                     IN  aay freq,
                                     IN  aay shift_a,
                                     IN  aay shift_b,
                                     IN  b   arm,
                                     IN  b   wait_for_arm_ack,
                                     OUT b   low_fill);
Flush                               ();
SetActiveFunctionGenerators         (IN  as  names);
ResetActiveFunctionGenerators       ();
ReadExecutedParameterCounts         (OUT au  result);
ReadFillLevels                      (OUT at  result);
ReadAllNames                        (OUT as  names);
ReadNames                           (OUT as  names);
ReadArmed                           (OUT ai  armed_states);
ReadEnabled                         (OUT ai  enabled_states);
ReadRunning                         (OUT ai  running_states);
```

Signals

```
Stopped       (s name,
               t time,
               b aborted,
               b hardwareMacroUnderflow,
               b microControllerUnderflow);
SigStopped    (s name,
```

```
                T time,
                b aborted,
                b hardwareMacroUnderflow,
                b microControllerUnderflow);
Armed          (s name,
                b armed);
Enabled        (s name,
                b enabled);
Running        (s name,
                b running);
Started        (s name,
                t time);
SigStarted     (s name,
                T time);
Refill         (s name);
AllStopped     (t time);
SigAllStopped (T time);
AllArmed       ();
```

Properties

```
StartTag                    readwrite  u
GenerateIndividualSignals   readwrite  b
```

Description

Operation of function generators is aggregated to reduce the number of d-bus operations required.

Method Details

The Arm() method

```
Arm ();
```

Enable all function generators that have data and arm them.

A function generator can only be Armed if FillLevel is non-zero. An Enabled function generator can not be Armed again until it either completes waveform generation or the user calls Abort. Arming a function generator takes time. Wait for Armed to transition to true before sending StartTag.

The Abort() method

```
Abort (IN  b wait_for_abort_ack);
```

Abort waveform generation in all function generators.

disabled state before returning.

This directs the hardware to stop waveform generation. If the function generator was Armed, it is disarmed and disabled, without outputting any waveform data. If the function generator is running, output is Stopped at the current value and disabled. Aborting a function generator takes time, so even after a call to Abort, the function generator might still be Started. However, it will reach the disabled state as quickly as it can, transitioning through Stopped as usual. If the Owner of a FunctionGenerator quits without running Disown, the Abort is run automatically.

'`IN b wait_for_abort_ack`':
: If true, wait for all FGs to transition to the

The InitializeSharedMemory() method

```
InitializeSharedMemory (IN  s shared_memory_name);
```

'`IN s shared_memory_name`':

The AppendParameterTuplesForBeamProcess() method

```
AppendParameterTuplesForBeamProcess (IN  i beam_process,
                                     IN  b arm,
                                     IN  b wait_for_arm_ack);
```

'`IN i beam_process`':
'`IN b arm`':
'`IN b wait_for_arm_ack`':

The AppendParameterSets() method

```
AppendParameterSets (IN  aan coeff_a,
                     IN  aan coeff_b,
                     IN  aai coeff_c,
                     IN  aay step,
                     IN  aay freq,
                     IN  aay shift_a,
                     IN  aay shift_b,
```

```
IN  b   arm,
IN  b   wait_for_arm_ack,
OUT b   low_fill);
```

For each function generator, append parameter tuples describing

the waveform to generate. Each parameter is sent as a vector of vectors: per FG then per tuple element

Parameters are sent as a vector of vectors. The outside vectors contain a coefficient vector for each FG and must be of the same size and less or equal the number of active FGs. The coefficient vectors for each FG's parameter set must be the same size but different FGs may have different parameter set sizes. If there is no data for an individual function generator an empty vector should be sent.

`coeff_a`: Quadratic coefficient (a), 16-bit signed `coeff_b`: Linear coefficient (b), 16-bit signed `coeff_c`: Constant coefficient (c), 32-bit signed `step`: Number of interpolated samples (0=250, 1=500, 2=1000, 3=2000, 4=4000, 5=8000, 6=16000, or 7=32000) `freq`: Output sample frequency (0=16kHz, 1=32kHz, 2=64kHz, 3=125kHz, 4=250kHz, 5=500kHz, 6=1MHz, or 7=2MHz) `shift_a`: Exponent of coeff_a, 6-bit unsigned; a = coeff_a*2^shift_a `shift_b`: Exponent of coeff_b, 6-bit unsigned; b = coeff_b*2^shift_b

`arm`: If true, arm each function generator that received data and wait for acknowledgement `wait_for_arm_ack`: If true, and arm is true, wait for arm acknowledgements from all fgs before returning `low_fill`: Fill level remains low for at least one FG - use ReadFillLevels

This function appends the parameter vectors (which must be equal in length) to the FIFO of remaining waveform to generate. Each parameter set (coefficients) describes a number of output samples in the generated wave form. Parameter sets are executed in order until no more remain.

If the fill level is not high enough, this method returns true. Only once this function has returned false can you await the Refill signal.

At each step, the function generator outputs high_bits(c*2^32 + b*t + c*t*t), where t ranges from 0 to numSteps-1. high_bits are the high OutputWindowSize bits of the resulting 64-bit signed value.

'`IN aan coeff_a`':
'`IN aan coeff_b`':
'`IN aai coeff_c`':
'`IN aay step`':
'`IN aay freq`':
'`IN aay shift_a`':
'`IN aay shift_b`':
'`IN b arm`':
'`IN b wait_for_arm_ack`':
'`OUT b low_fill`':

The Flush() method

```
Flush ();
```

Empty the parameter tuple set of all function generators.

Flush may only be called when not Enabled. Flush does not clear the ExecutedParameterCount.

The SetActiveFunctionGenerators() method

```
SetActiveFunctionGenerators (IN  as names);
```

Set the list of active function generators handled by this master to the names given. `names`: List of names identifying each FG to activate. Format: fg-[SCUBusSlot]-[DeviceNumber]-[index] e.g. {"fg-3-0-0","fg-3-1-1"}

'`IN as names`':

The ResetActiveFunctionGenerators() method

```
ResetActiveFunctionGenerators ();
```

Resets the list of active function generators handled by this master to all available FGs.

The ReadExecutedParameterCounts() method

```
ReadExecutedParameterCounts (OUT au result);
```

Number of parameter tuples executed by each function generator

This counts the total number of parameter tuples executed since the last Started signal. Obviously, if the function generator is running, the returned value will be old.

'`OUT au result`':

Number tuples executed by hardware.

The ReadFillLevels() method

```
ReadFillLevels (OUT at result);
```

Remaining waveform data in nanoseconds for each FG.

The SAFTd has sufficient parameters buffered to supply the function generator with data for the specified time in nanoseconds. Note, due to the slow nature of software, if

the function generator is currently running, the read value will already be out-of-date upon return. This property should be used for informational use only.

'`OUT at result`':
: Remaining waveform data in nanoseconds for each FG.

The ReadAllNames() method

```
ReadAllNames (OUT as names);
```

'`OUT as names`':

The ReadNames() method

```
ReadNames (OUT as names);
```

Read the name for each active FG.

'`OUT as names`':
: Name of each FG.

The ReadArmed() method

```
ReadArmed (OUT ai armed_states);
```

Read the armed state of each active FG.

'`OUT ai armed_states`':
: State of each FG.

The ReadEnabled() method

```
ReadEnabled (OUT ai enabled_states);
```

Read the enabled state of each active FG.

'`OUT ai enabled_states`':
: State of each FG.

The ReadRunning() method

```
ReadRunning (OUT ai running_states);
```

Read the running state of each active FG.

'`OUT ai running_states`':
: State of each FG.

Signal Details

The "Stopped" signal

```
Stopped (s name,
         t time,
         b aborted,
         b hardwareMacroUnderflow,
         b microControllerUnderflow);
```

Deprecated! Use SigStopped instead.

'`s name`': Name of the FG that generated the signal.

'`t time`': Time when function generation ended in nanoseconds since 1970

'`b aborted`':
'`b hardwareMacroUnderflow`':
: A fatal error, indicating the SCUbus is congested

'`b microControllerUnderflow`':
: A fatal error, indicating the host CPU is overloaded

The "SigStopped" signal

```
SigStopped (s name,
            T time,
            b aborted,
            b hardwareMacroUnderflow,
            b microControllerUnderflow);
```

Stopped signal forwarded from a single Function Generator

The function generator stops either successfully (when all data has been sent), or it stops due to an error. When an error occurs, the function generator stops and holds its most recent value. This can occur due to two causes:

hardwareMacroUnderflow, a fatal error indicating the hardware ran out of data. If the SCUbus is too busy, it can happen that the waveform data stored in the function generator HDL is not refilled in time. This error can only be mitigated by ensuring that the function generator does not share the SCUbus with other users.

microControllerUnderflow, a fatal error indicating the microcontroller ran out of data. If the host CPU running this software is too busy, it can happen that the waveform data is not delivered to the microcontroller before the microcontroller runs out of data. This error can be mitigated by reducing the number of busy processes running on the system.

Once the function generator has stopped, ExecutedParameterCount remains valid until the next time the function generator starts. After stopping, regardless of if the generation was successful or not, the parameter FIFO is cleared, Enabled is false, and this signal emitted.

'`s name`': Name of the FG that generated the signal.

'`T time`': Time when function generation ended in nanoseconds since 1970

'`b aborted`':
'`b hardwareMacroUnderflow`':
A fatal error, indicating the SCUbus is congested

'`b microControllerUnderflow`':
A fatal error, indicating the host CPU is overloaded

The "Armed" signal

```
Armed (s name,
       b armed);
```

Armed signal forwarded from a single Function Generator

'`s name`':

'`b armed`':

The "Enabled" signal

```
Enabled (s name,
         b enabled);
```

Enabled signal forwarded from a single Function Generator

'`s name`':

'`b enabled`':

The "Running" signal

```
Running (s name,
         b running);
```

Running signal forwarded from a single Function Generator

'`s name`':

'`b running`':

The "Started" signal

```
Started (s name,
         t time);
```

Deprecated! Use SigStarted instead.

'`s name`':

'`t time`':

The "SigStarted" signal

```
SigStarted (s name,
            T time);
```

Started signal forwarded from a single Function Generator

'`s name`':

'`T time`':

The "Refill" signal

```
Refill (s name);
```

Refill signal forwarded from a single Function Generator

'`s name`':

The "AllStopped" signal

```
AllStopped (t time);
```

Deprecated! Use SigAllStopped instead.

This signal is generated when a function generator stops and all function generators controlled by this master have then stopped.

'`t time`': Time when last function generation ended in nanoseconds since 1970

The "SigAllStopped" signal

```
SigAllStopped (T time);
```

All Function generators have stopped.

This signal is generated when a function generator stops and all function generators controlled by this master have then stopped.

'`T time`': Time when last function generation ended

The "AllArmed" signal

```
AllArmed ();
```

All Function generators that have a FillLevel>0 are armed.

This signal is generated when a function generator signals that it is armed, and all function generators controlled by this master that have FillLevel>0 are armed.

Property Details

The "StartTag" property

```
StartTag  readwrite  u
```

The SCUbus tag which causes function generation to begin.

All function generators under control of the Master use the same tag. If the function generator is Armed and this tag is sent to the SCUbus, then the function generator will begin generating the output waveform. StartTag may only be set when the FunctionGenerator is not Enabled.

The "GenerateIndividualSignals" property

```
GenerateIndividualSignals  readwrite  b
```

If true, signals from the individual

function generators are forwarded. The aggregate signals will still be generated. This defaults to false to reduce the load on the d-bus message bus.

de.gsi.saftlib.OutputActionSink

Name

de.gsi.saftlib.OutputActionSink — An output through which on/off actions flow.

Methods

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  b on,
              OUT s result);
WriteOutput  (IN  b value);
ReadOutput   (OUT b value);
StartClock   (IN  d high_phase,
              IN  d low_phase,
              IN  t phase_offset,
              OUT b running);
StopClock    (OUT b running);
```

Properties

```
OutputEnable                readwrite  b
SpecialPurposeOut           readwrite  b
GateOut                     readwrite  b
BuTiSMultiplexer            readwrite  b
PPSMultiplexer              readwrite  b
OutputEnableAvailable       readable   b
SpecialPurposeOutAvailable  readable   b
LogicLevelOut               readable   s
IndexOut                    readable   u
TypeOut                     readable   s
Input                       readable   s
```

Description

This inteface allows the generation of Output pulses. An OutputActionSink is also an ActionSink and Owned.

If two SoftwareConditions are created on the same SoftwareActionSink which require simultaneous delivery of two Actions, then they will be delivered in arbitrary order, both having the 'conflict' flag set.

Method Details

The NewCondition() method

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  b on,
              OUT s result);
```

Create a condition to match incoming events

This method creates a new condition that matches events whose identifier lies in the range [id & mask, id | ~mask]. The offset acts as a delay which is added to the event's execution timestamp to determine the timestamp when the matching condition fires its action. The returned object path is a OutputCondition object.

'`IN b active`':
: Should the condition be immediately active

'`IN t id`': Event ID to match incoming event IDs against

'`IN t mask`':
: Set of bits for which the event ID and id must agree

'`IN x offset`':
: Delay in nanoseconds between event and action

'`IN b on`': The output should be toggled on (or off)

'`OUT s result`':
: Object path to the created OutputCondition

The WriteOutput() method

```
WriteOutput (IN  b value);
```

Directly manipulate the output value.

Set the output to on/off. Overwrite the previous state, regardless of whether it came from WriteOutput or a matching Condition. Similarly, the value may in turn be overwritten by a subsequent matching Condition or WriteOutput.

'`IN b value`':

The ReadOutput() method

```
ReadOutput (OUT b value);
```

Read the output state.

This property reflects the current value which would be output when OutputEnable is true. This may differ from ReadInput on an inout.

'`OUT b value`':

The StartClock() method

```
StartClock (IN  d high_phase,
            IN  d low_phase,
            IN  t phase_offset,
            OUT b running);
```

Starts the clock generator with given parameters.

All parameters expect the value in nanoseconds.

'`IN d high_phase`':
'`IN d low_phase`':
'`IN t phase_offset`':
'`OUT b running`':

The StopClock() method

```
StopClock (OUT b running);
```

Stops the clock generator.

'`OUT b running`':

Property Details

The "OutputEnable" property

```
OutputEnable  readwrite  b
```

Is the output driver enabled.

When OutputEnable is false, the output is not driven. This defaults to off. See also Termination if this is an inoutput.

The "SpecialPurposeOut" property

```
SpecialPurposeOut  readwrite  b
```

Is the special function enabled.

The "GateOut" property

```
GateOut  readwrite  b
```

Set output gate or get gate status.

The "BuTiSMultiplexer" property

```
BuTiSMultiplexer  readwrite  b
```

Output BuTiS t0 with timestamp.

The "PPSMultiplexer" property

```
PPSMultiplexer  readwrite  b
```

Output PPS signal from White Rabbit core.

The "OutputEnableAvailable" property

```
OutputEnableAvailable  readable   b
```

Can output enable be configured.

The "SpecialPurposeOutAvailable" property

```
SpecialPurposeOutAvailable  readable   b
```

Can special configuration be configured.

The "LogicLevelOut" property

```
LogicLevelOut  readable   s
```

Logic level of the output (LVDS, LVTTL, ...)

The "IndexOut" property

```
IndexOut  readable   u
```

IO index.

The "TypeOut" property

```
TypeOut  readable   s
```

IO type (GPIO, LVDS, ...)

The "Input" property

```
Input  readable   s
```

If non-empty, path of the Input object for the same physical IO

de.gsi.saftlib.OutputCondition

Name

de.gsi.saftlib.OutputCondition — Matched against incoming events on an OutputActionSink.

Properties

```
On  readwrite  b
```

Description

OutputConditions are created by OutputActionSinks to select which events should generate signal toggles. This interface always implies that the object also implements the general Condition interface.

Property Details

The "On" property

```
On  readwrite  b
```

Should the Output be turned on or off

de.gsi.saftlib.Owned

Name

de.gsi.saftlib.Owned — A DBus object which can grant exclusive access.

Methods

```
Disown  ();
Own     ();
Destroy ();
```

Signals

```
Destroyed ();
```

Properties

```
Owner         readable   s
Destructible  readable   b
```

Description

This interface allows clients to claim ownership of the object. When the object has no owner, full access is granted to all clients. When owned, only the owner may execute priveledged methods. If the object is Destructible and has an Owner, the object will be automatically Destroyed when the Owner quits.

Method Details

The Disown() method

```
Disown ();
```

Release ownership of the object.

This method may only be invoked by the current owner of the object. A disowned object may be accessed by all clients and will persist until its Destroy method is called.

The Own() method

```
Own ();
```

Claim ownership of the object.

This method may only be invoked if the object is unowned.

The Destroy() method

```
Destroy ();
```

Destroy this object.

This method may only be invoked by the current owner of the object. However, if the condition has been disowned, it may be invoked by anyone. reset_connection="false" prevents the DBus service from resetting the connection of a destroyed object.

Signal Details

The "Destroyed" signal

```
Destroyed ();
```

The object was destroyed.

Property Details

The "Owner" property

```
Owner  readable   s
```

The dbus client which owns this object.

If there is no Owners, the empty string is returned. Only the owner may access privileged methods on the object. When the owning client disconnects, ownership will be automatically released, and if the object is Destructable, the object will also be automatically Destroyed.

The "Destructible" property

```
Destructible  readable   b
```

Can the object be destroyed.

A destructible object represents a temporary allocated resource. When the owner quits, the object will be automatically Destoyed. Some objects are indestructible, representing a physical resource.

de.gsi.saftlib.SAFTd

Name

de.gsi.saftlib.SAFTd — Report instances of managed hardware

Methods

```
AttachDevice (IN  s name,
              IN  s path,
              OUT s result);
RemoveDevice (IN  s name);
Quit         ();
```

Properties

```
SourceVersion  readable   s
BuildInfo      readable   s
Devices        readable   a{ss}
```

Description

This D-Bus interface is the main entry point into the saftlib. All device drivers register their top-level hardware objects here.

Method Details

The AttachDevice() method

```
AttachDevice (IN  s name,
              IN  s path,
              OUT s result);
```

Instruct saftd to control a new device.

Devices are attached to saftlib by specifying a name and a path. The name should denote the logical relationship of the device to saftd. For example, baseboard would be a good name for the timing receiver attached to an SCU. If an exploder is being used to output events to an oscilloscope, a good logical name might be scope. In these examples,

the path for the SCU baseboard would be dev/wbm0, and the scope exploder would be dev/ttyUSB3 or similar.

This scheme is intended to make it easy to hot-swap hardware. If the exploder dies, you can simply attach a new one under the same logical name, even though the path might be different.

'`IN s name`':
: The logical name for the device

'`IN s path`':
: The etherbone path where the device can be found

'`OUT s result`':
: Object path of the created device

The RemoveDevice() method

```
RemoveDevice (IN  s name);
```

Remove the device from saftlib management.

'`IN s name`':
: The logical name for the device

The Quit() method

```
Quit ();
```

Instructs the saftlib daemon to quit.

Be absolutely certain before calling this method! All clients will have their future calls throw exceptions.

Property Details

The "SourceVersion" property

```
SourceVersion  readable   s
```

SAFTd source version.

The version of the SAFTd source code this daemon was compiled from. Format is "saftlib #.#.# (git-id): MMM DD YYYY HH:MM:SS".

The "BuildInfo" property

```
BuildInfo  readable   s
```

SAFTd build information.

Information about when and where the SAFTd was compiled. Format is "built by USERNAME on MMM DD YYYY HH:MM:SS with HOSTNAME running OPERATING-SYSTEM".

The "Devices" property

```
Devices  readable   a{ss}
```

List of all devices attached to saftd.

The key is the name of the device as provided to AttachDevice. The value is the dbus path to the Device object, NOT the etherbone path of the device. Each object is guaranteed to implement at least the Device interface, however, typically the objects implement the TimingReceiver interface.

de.gsi.saftlib.SCUbusActionSink

Name

de.gsi.saftlib.SCUbusActionSink — An output through which SCUbus actions flow.

Methods

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  u tag,
              OUT s result);
InjectTag    (IN  u tag);
```

Description

This inteface allows the generation of SCU timing events. A SCUbusActionSink is also an ActionSink and Owned.

If two SoftwareConditions are created on the same SoftwareActionSink which require simultaneous delivery of two Actions, then they will be delivered in arbitrary order, both having the 'conflict' flag set.

Method Details

The NewCondition() method

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              IN  u tag,
              OUT s result);
```

Create a condition to match incoming events

This method creates a new condition that matches events whose identifier lies in the range [id & mask, id | ~mask]. The offset acts as a delay which is added to the event's

execution timestamp to determine the timestamp when the matching condition fires its action. The returned object path is a SCUBUSCondition object.

'`IN b active`':
: Should the condition be immediately active

'`IN t id`': Event ID to match incoming event IDs against

'`IN t mask`':
: Set of bits for which the event ID and id must agree

'`IN x offset`':
: Delay in nanoseconds between event and action

'`IN u tag`':
: The 32-bit value to send on the SCUbus

'`OUT s result`':
: Object path to the created SCUbusCondition

The InjectTag() method

```
InjectTag (IN  u tag);
```

Directly generate a SCUbus timing event.

For debugging, it can be helpful to simply create SCUbus events without a matching timing event.

'`IN u tag`':
: The 32-bit value to push to the SCUbus.

de.gsi.saftlib.SCUbusCondition

Name

de.gsi.saftlib.SCUbusCondition — Matched against incoming events on a SCUbusActionSink.

Properties

```
Tag  readwrite  u
```

Description

SCUbusConditions are created by SCUbusActionSinks to select which events should generate callbacks. This interface always implies that the object also implements the general Condition interface.

Property Details

The "Tag" property

```
Tag  readwrite  u
```

The tag which is sent to the SCUbus by this condition

de.gsi.saftlib.SoftwareActionSink

Name

de.gsi.saftlib.SoftwareActionSink — An output through which software actions flow.

Methods

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              OUT s result);
```

Description

A SoftwareActionSink guarantees ordered execution of all callbacks on SoftwareConditions created via the NewCondition method. Each SoftwareActionSink is independent of all others, so a single program may operate in isolation without concern about potential conflicting rules from other clients on the same machine.

A SoftwareActionSink is both an ActionSink and Owned.

If two SoftwareConditions are created on the same SoftwareActionSink which require simultaneous delivery of two Actions, then they will be delivered in arbitrary order, both having the 'conflict' flag set.

Method Details

The NewCondition() method

```
NewCondition (IN  b active,
              IN  t id,
              IN  t mask,
              IN  x offset,
              OUT s result);
```

Create a condition to match incoming events

This method creates a new condition that matches events whose identifier lies in the range [id & mask, id | ~mask]. The offset acts as a delay which is added to the event's

execution timestamp to determine the timestamp when the matching condition fires its action. The returned object path is a SoftwareCondition object.

'`IN b active`':
: Should the condition be immediately active

'`IN t id`': Event ID to match incoming event IDs against

'`IN t mask`':
: Set of bits for which the event ID and id must agree

'`IN x offset`':
: Delay in nanoseconds between event and action

'`OUT s result`':
: Object path to the created SoftwareCondition

de.gsi.saftlib.SoftwareCondition

Name

de.gsi.saftlib.SoftwareCondition — Matched against incoming events on a SoftwareActionSink.

Signals

```
Action    (t event,
           t param,
           t deadline,
           t executed,
           q flags);
SigAction (t event,
           t param,
           T deadline,
           T executed,
           q flags);
```

Description

SoftwareConditions are created by SoftwareActionSinks to select which events should generate callbacks. This interface always implies that the object also implements the general Condition interface.

Signal Details

The "Action" signal

```
Action (t event,
        t param,
        t deadline,
        t executed,
        q flags);
```

Deprecated! Use SigAction instead.

‘`t event`’: The event identifier that matched this rule.

‘`t param`’: The parameter field, whose meaning depends on the event ID.

‘`t deadline`’:
: The scheduled execution timestamp of the action (event time + offset).

‘`t executed`’:
: The actual execution timestamp of the action.

‘`q flags`’: Whether the action was (ok=0,late=1,early=2,conflict=4,delayed=8)

The "SigAction" signal

```
SigAction (t event,
           t param,
           T deadline,
           T executed,
           q flags);
```

Emitted whenever the condition matches a timing event.

While the underlying hardware strives to deliver the action precisely on the deadline, the software stack adds non-deterministic delay, so the deadline may be milliseconds in the past. The late flag only indicates that the hardware failed to meet the required deadline. Similarly, the executed timestamp is the time when the hardware delivered the action, not when software has received it.

Actions with error flags (late, early, conflict, delayed) are only delivered to this signal if the Condition which generated them specified that the respective error should be accepted.

‘`t event`’: The event identifier that matched this rule.

‘`t param`’: The parameter field, whose meaning depends on the event ID.

‘`T deadline`’:
: The scheduled execution timestamp of the action (event time + offset).

‘`T executed`’:
: The actual execution timestamp of the action.

‘`q flags`’: Whether the action was (ok=0,late=1,early=2,conflict=4,delayed=8)

de.gsi.saftlib.TimingReceiver

Name

de.gsi.saftlib.TimingReceiver — A timing receiver.

Methods

```
CurrentTemperature    (OUT i result);
ReadCurrentTime       (OUT t result);
CurrentTime           (OUT T result);
NewSoftwareActionSink (IN  s name,
                       OUT s result);
InjectEvent           (IN  t event,
                       IN  t param,
                       IN  t time);
InjectEvent           (IN  t event,
                       IN  t param,
                       IN  T time);
```

Signals

```
SigLocked (b locked);
```

Properties

```
GatewareInfo            readable   a{ss}
GatewareVersion         readable   s
Locked                  readable   b
TemperatureSensorAvail  readable   b
SoftwareActionSinks     readable   a{ss}
Outputs                 readable   a{ss}
Inputs                  readable   a{ss}
Interfaces              readable   a{sa{ss}}
Free                    readable   u
```

Description

Timing receivers can respond to timing events from the data master. The can also respond to external timing triggers via inputs.

The general idea is that a TimingReceiver has ActionSinks to which it sends actions in response to incoming timing events. Timing events are matched with Conditions to create the Actions sent to the Sinks.

EventSources are objects which create timing events, to be matched by the conditions. The data master is a global EventSource to which all TimingReceivers listen. However, external inputs can also be configured to generate timing events. Furthermore, a TimingReceiver can simulate the receipt of a timing event by calling the InjectEvent method.

Timing receivers always typically have binary outputs lines (OutputActionSinks), which are listed in the Outputs property. Similarly, they often have digital inputs (InputEventSources).

Some timing receivers have special purpose interfaces. For example, an SCU has the SCUbusActionSink which generates 32-bit messages over the SCU backplane. These special interfaces can be found in the interfaces property. The SCU backplane would be found under the SCUbusActionSink key, and as there is only one, it would be the 0th.

Method Details

The CurrentTemperature() method

```
CurrentTemperature (OUT i result);
```

The current temperature in degree Celsius.

The valid temperature range is from -70 to 127 degree Celsius. The data type is 32-bit signed integer.

'`OUT i result`':
: Temperature in degree Celsius.

The ReadCurrentTime() method

```
ReadCurrentTime (OUT t result);
```

This method is deprecaded, use CurrentTime instead.

The current time in nanoseconds since 1970. Due to delays in software, the returned value is probably several milliseconds behind the true time.

'`OUT t result`':
: Nanoseconds since 1970.

The CurrentTime() method

```
CurrentTime (OUT T result);
```

The current time of the timingreceiver.

The result type is saftlib::Time, which can be used to obtain either the number of nanoseconds since 1970, or the same value minus the current UTC offset. Due to delays in software, the returned value is probably several milliseconds behind the true time.

'`OUT T result`':
the current time of the timingreceiver

The NewSoftwareActionSink() method

```
NewSoftwareActionSink (IN  s name,
                       OUT s result);
```

Create a new SoftwareActionSink.

SoftwareActionSinks allow a program to create conditions that match incoming timing events. These conditions may have callback methods attached to them in order to receive notification. The returned path corresponds to a SoftwareActionSink that is owned by the process which claimed it, and can thus be certain that no other processes can interfere with the results.

'`IN s name`':
A name for the SoftwareActionSink. Can be left blank.

'`OUT s result`':
Object path to the created SoftwareActionSink.

The InjectEvent() method

```
InjectEvent (IN  t event,
             IN  t param,
             IN  t time);
```

The method is depricated, use InjectEvent with saftlib::Time instead.

Simulate the receipt of a timing event Sometimes it is useful to simulate the receipt of a timing event. This allows software to test that configured conditions lead to the desired behaviour without needing the data master to send anything.

'`IN t event`':
The event identifier which is matched against Conditions

'`IN t param`':
The parameter field, whose meaning depends on the event ID.

‘IN t time’:
The execution time for the event, added to condition offsets.

The InjectEvent() method

```
InjectEvent (IN  t event,
             IN  t param,
             IN  T time);
```

Simulate the receipt of a timing event

event: The event identifier which is matched against Conditions param: The parameter field, whose meaning depends on the event ID. time: The execution time for the event, added to condition offsets.

Sometimes it is useful to simulate the receipt of a timing event. This allows software to test that configured conditions lead to the desired behaviour without needing the data master to send anything.

‘IN t event’:
‘IN t param’:
‘IN T time’:

Signal Details

The "SigLocked" signal

```
SigLocked (b locked);
```

‘b locked’:

Property Details

The "GatewareInfo" property

```
GatewareInfo  readable   a{ss}
```

Key-value map of hardware build information

The "GatewareVersion" property

```
GatewareVersion  readable   s
```

Hardware build version

Returns "major.minor.tiny" if version is valid (or "N/A" if not available)

The "Locked" property

```
Locked  readable   b
```

The timing receiver is locked to the timing grandmaster.

Upon power-up it takes approximately one minute until the timing receiver has a correct timestamp.

The "TemperatureSensorAvail" property

```
TemperatureSensorAvail  readable   b
```

Check if a temperature sensor is available

in a timing receiver.

The "SoftwareActionSinks" property

```
SoftwareActionSinks  readable   a{ss}
```

A list of all current SoftwareActionSinks.

Typically, these SoftwareActionSinks will be owned by their processes and not of much interest to others. Therefore, many of the entries here may be of no interest to a particular client. However, it is possible for a SoftwareActionSink to be Disowned, in which case it may be persistent and shared between programs under a well known name.

The "Outputs" property

```
Outputs  readable   a{ss}
```

A list of all the high/low outputs on the receiver.

Each path refers to an object of type Output.

The "Inputs" property

```
Inputs  readable   a{ss}
```

A list of all the high/low inputs on the receiver.

Each path refers to an object of type Input.

The "Interfaces" property

```
Interfaces  readable   a{sa{ss}}
```

List of all object instances of various hardware.

The key in the dictionary is the name of the interface. The value is all object paths to hardware implementing that interface.

The "Free" property

```
Free  readable   u
```

The number of additional conditions that may be activated.

The ECA has limited hardware resources in its match table.

de.gsi.saftlib.WrMilGateway

Name

de.gsi.saftlib.WrMilGateway — A converter for WhiteRabbit timing events to MIL.

Methods

```
StartSIS18      ();
StartESR        ();
ClearStatistics ();
ResetGateway    ();
KillGateway     ();
```

Signals

```
SigFirmwareState    (u state);
SigFirmwareRunning  (b running);
SigEventSource      (u source);
SigNumLateMilEvents (u total,
                     u since_last_signal);
SigInUse            (b inUse);
```

Properties

```
RegisterContent   readable   au
MilHistogram      readable   au
WrMilMagic        readable   u
FirmwareState     readable   u
FirmwareRunning   readable   b
EventSource       readable   u
UtcTrigger        readwrite  y
EventLatency      readwrite  u
UtcUtcDelay       readwrite  u
TriggerUtcDelay   readwrite  u
UtcOffset         readwrite  t
NumMilEvents      readable   t
LateHistogram     readable   au
NumLateMilEvents  readable   u
InUse             readable   b
```

Description

The WhiteRabbit-MIL-Gateway receives WhiteRabbit timing events, translates them into MIL timing events and outputs them with few microsecond accuracy. This device is needed to run SIS18 and ESR.

Method Details

The StartSIS18() method

```
StartSIS18 ();
```

Configure Gateway as SIS18 "Pulszentrale" and start

The StartESR() method

```
StartESR ();
```

Configure Gateway as ESR "Pulszentrale" and start

The ClearStatistics() method

```
ClearStatistics ();
```

Reset all event counters and histograms

The ResetGateway() method

```
ResetGateway ();
```

Stop WR-MIL Gateway and restart after 1 second pause

The KillGateway() method

```
KillGateway ();
```

Kill WR-MIL Gateway. Only MCU reset can recover (useful to do before loading new firmware)

Signal Details

The "SigFirmwareState" signal

```
SigFirmwareState (u state);
```

signal emitted if firmware state changes

‘`u state`’:

The "SigFirmwareRunning" signal

```
SigFirmwareRunning (b running);
```

signal emitted if starts/stops running

‘`b running`’:

The "SigEventSource" signal

```
SigEventSource (u source);
```

signal emitted if event source changes

‘`u source`’:

The "SigNumLateMilEvents" signal

```
SigNumLateMilEvents (u total,
                     u since_last_signal);
```

signal emitted if number of late events increases

‘`u total`’:

‘`u since_last_signal`’:

The "SigInUse" signal

```
SigInUse (b inUse);
```

signal emitted if number of events increases

'`b inUse`':

Property Details

The "RegisterContent" property

```
RegisterContent  readable   au
```

Access all registers of the WrMilGateway

The "MilHistogram" property

```
MilHistogram  readable   au
```

Access MIL event histogram WrMilGateway

The "WrMilMagic" property

```
WrMilMagic  readable   u
```

WR-MIL magic number

The "FirmwareState" property

```
FirmwareState  readable   u
```

WR-MIL firmware state

The "FirmwareRunning" property

```
FirmwareRunning  readable   b
```

WR-MIL firmware is active (not stalled)

The "EventSource" property

```
EventSource  readable  u
```

WR-MIL event source.

This can be:

0 -> not configured (default)

1 -> SIS18 (work as SIS18 "Pulszentrale")

2 -> ESR (work as ESR "Pulszentrale")

The "UtcTrigger" property

```
UtcTrigger  readwrite  y
```

The event number that triggers generation of UTC events

The "EventLatency" property

```
EventLatency  readwrite  u
```

Event latency in microseconds.

This is the time between the WR-event deadline and the rising edge at the TIF module caused by the translated MIL event

The "UtcUtcDelay" property

```
UtcUtcDelay  readwrite  u
```

Time between the generated UTC MIL events

The "TriggerUtcDelay" property

```
TriggerUtcDelay  readwrite  u
```

Time between the UTC trigger event and the first UTC MIL event

The "UtcOffset" property

```
UtcOffset  readwrite  t
```

seconds between 01.01.2008 and 01.01.1970

The number of seconds that is added to the WR-TAI (which starts at 1970) to create a UTC time that starts at 2008 (64bit value)

The "NumMilEvents" property

```
NumMilEvents  readable  t
```

The number of translated events (as 64bit value)

The "LateHistogram" property

```
LateHistogram  readable  au
```

read a histogram of event delays

Bin content:

[0]: delay less than 2^10 ns

[1]: delay less than 2^11 ns

[2]: delay less than 2^12 ns

...

[14]: delay less than 2^24 ns

[15]: delay less than 2^25 ns

The "NumLateMilEvents" property

```
NumLateMilEvents  readable  u
```

The number of translated events that were submitted later
than the anticipated WR event deadline

The "InUse" property

```
InUse  readable  b
```

indicates if the gateway is translating any events