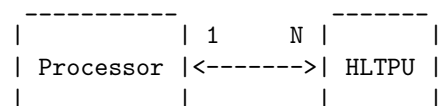


Description of Processor instance

The Processor implemented in the `Processor.h` and `Processor.cxx` files is an abstract base class. It is intended to be derived into the `DummyProcessor` and the `HltpuProcessor` classes. The former simulates processing. The later handles connections with the real HLTPU processes.

The Processor class has been designed so that it implements the common business logic. The Derived classes need only to implement the specific actions.

Data structure architecture



`Processor` is a class and `HLTPU` is a struct defined inside the `Processor` class. The `Processor` class contains

- a state information
- references to the `L1Source`, `DataCollector` and `EventBuilder` instances
- some time points, statistics and histograms
- a vector of pointers to `HLTPU` structs

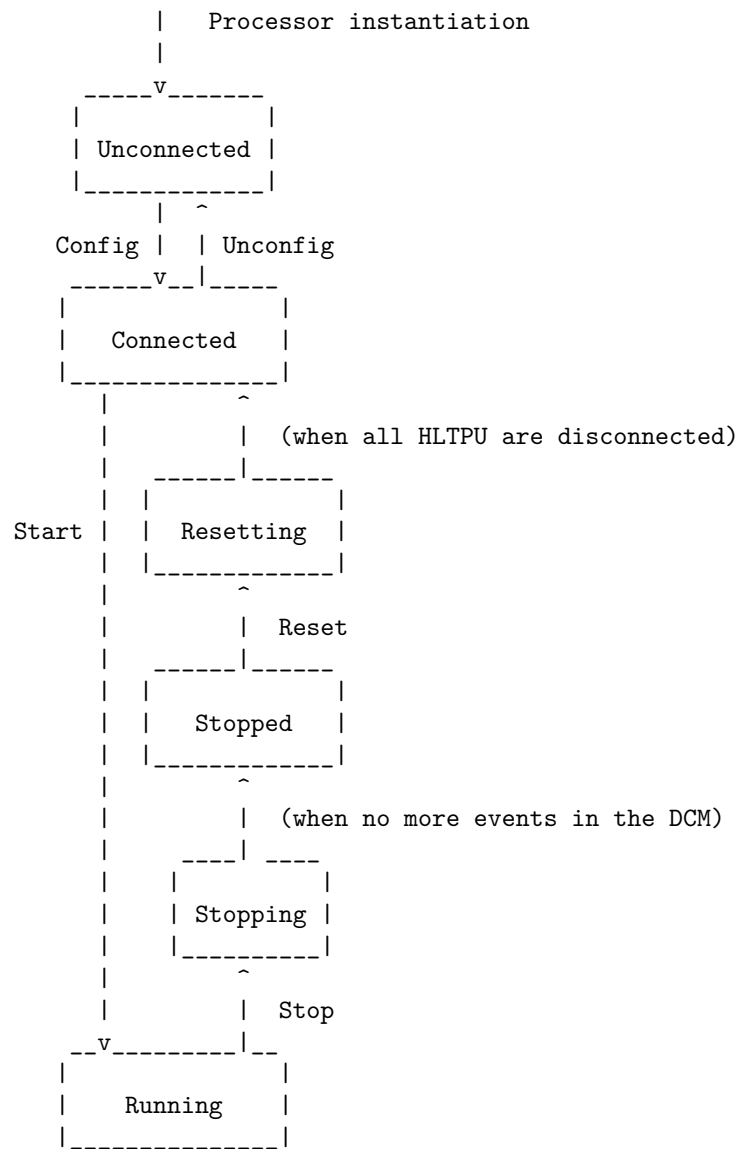
There is one `HLTPU` struct instance per `HLTPU` instance. The `HLTPU` struct contains

- the `HLTPU` name
- a time point (end of `IDLE` or `PROCESSING` time)
- the currently processed event
- some state information
- a pointer to the `Processor` instance

Execution logic

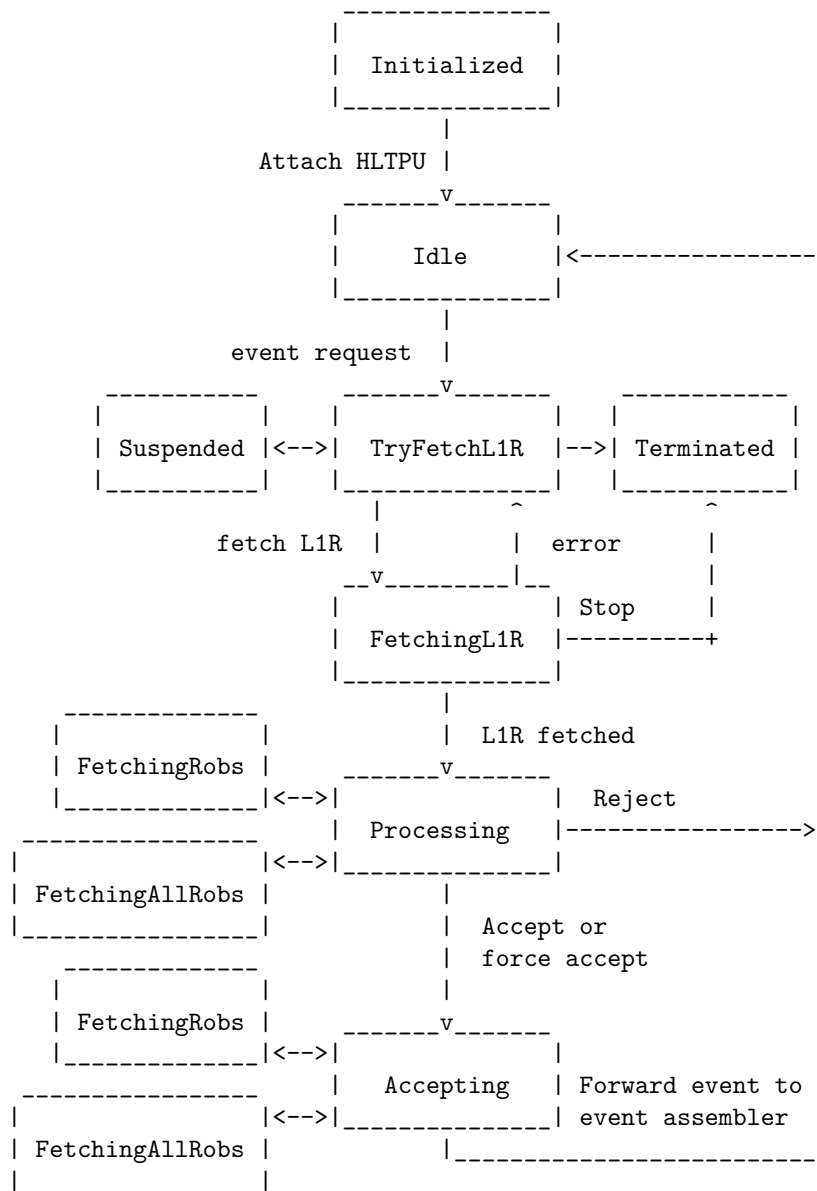
Processor state transition diagram

The `Processor` states match the run control states relatively closely. When the run control requests a stop, the `Processor` states goes from `Running` to `Connected`. The intermediate `Reset` transition allows faster stopping since the `EventBuilder` can be stopped once the `Processor` is stopped.



The state transitions is driven by the main routine following directives of the run controller.

HLTPU state transition diagram



HLTPU attachment will be rejected if the **Processor** is not in the **Connected** or **Running** state, or if the maximum connected HLTPU is reached.

When an event is requested, the HLTPU may get into the **Suspended** state when the **Processor** is not in the **Running** state, or if the backpressure quicked in.

The HLTPU leaves the **Suspended** state when the **Processor** is started, or the backpressure is released.

When the **Processor** is requested to stop and the HLTPU is in the **TryFetchL1R** or **FetchingL1R** states, the HLTPU is detached from the processor and goes into the **Terminated** state.

Code architecture

The code is designed so that all the common business logic may be contained in the **Processor** class. The **DummyProcessor** and **HltpuProcessor** contains only their specific tasks.

HLTPU struct creation

When an HLTPU struct is created, it is in the **Initialized** state. The **DummyProcessor** will automatically create the required number of HLTPU structs when the **Processor** starts. The **HltpuProcessor** will create the structs when a HLTPU process connects to the DCM.

An attempt to attach the HLTPU is then performed. It will be rejected if the **Processor** is not in the **Connected** or **Running** state, or if the maximum number of connected HLTPU has already been reached.

If it succeed, the method **setStateIdle()** is called.

setStateIdle()

This method moves the HLTPU into the **Idle** state. The method is abstract because the **DummyProcessor** and the **HltpuProcessor** do different operations.

The **DummyProcessor** does nothing beside changing the state and directly calls **tryFetchL1R()**. The **HltpuProcessor** starts waiting for an event request message from the HLTPU Processor. When this message is received, **tryFetchL1R()** is called.

tryFetchL1R()

The action of this method will depend on the state of the **Processor**.

If the **Processor** state is **Connected**, the HLTPU process connected and issued an event request before the **Processor** was started. The HLTPU is then put into **Suspended** state.

If the **Processor** state is **Stopping** or **Stopped**, the method **noMoreEvents()** is called. It notifies the HLTPU process that it must terminate.

If the state is **Running**, the action will depend on the backpressure. If the backpressure is active, the HLTPU will be put into **Suspended** state. Otherwise the `fetchL1R()` is called. The backpressure is active when the number of events accepted and not yet sent out yet is equal or above a maximum limit.

This method is not allowed to be called in any other **Processor** state. An assert failure will be triggered in that case.

unsuspendAllHltpu()

This method calls `tryFetchL1R()` for all suspended Hltpu. It is called when the **Processor** is started or stopped, or when the deletion of an event (`finalizeEvent()`) releases the backpressure.

fetchL1R()

This method sets the HLTPU state into **FetchingL1R**, creates the **event** instance and forwards the call to the **L1Source**. On completion, the callback method `onFetchL1R()` or `onFetchL1RError()` is called.

onFetchL1R()

Callback called when a L1R is received from the HLTSV. The Hltpu state changes to **Processing** and the Hltpu is initialized for the event processing. Some stats is updated and time point are stored to compute processing time.

If the event was reassigned by the HLTSV, or some invalid data is detected it is forced accepted, otherwise the method `processL1R()` is called.

processL1R()

This method is abstract because the **DummyProcessor** and **HltpuProcessor** have a specific implementation. At this stage, the Hltpu is in the **Processing** state depicted in the state diagram above.

The **DummyProcessor** calls a method that will simulate a level 2 processing (prior gathering all robs). It will requests some robs by calling `fetchRobs()`.

The **HltpuProcessor** forwards the L1R to the HLTPU process and waits for a message from the HLTPU. When the *fetchRobs* message is received, the method `fetchRobs()` is called. When the *accept* message is received, some checking is performed. If something bogus is detected, `forceAccept()` is called. Otherwise the `accept()` method is called. When the *reject* message is received, the `reject()` method is called.

onFetchRobs()

Callback method called when fetching robs is completed. This method may be called in multiple conditions.

- while a forceAccept is in progress that requested to fetch all Robs. In this case we call `finalizeForecAccept()`.
- while an event is accepted and requires to fetch some Robs. The method `finalizeAccept()` is called and `setStateIdle()` after it to close the state transision loop.
- while an event is processed and robs where requested. The overridden method `processRobs()` is then called.

processRobs()

This method is called when the event is processed. It is overridden by the DummyProcessor and the HltpuProcessor to implement specific behavior.

The DummyProcessor will simulate processing of a random duration and randomly chose between requesting more robs, requesting all robs, accepting or rejecting the event. It will call the corresponding `fetchRobs()`, `accept()` or `reject()` method.

reject()

This method will call in sequerce the `finalizeProcessing()`, the `finalizeEvent()`, and the `setStateIdle()` methods. The later will close the loop of state transitions.

finalizeProcessing()

This method simply update stats and histograms. It is also called when accepting an event is completed.

finalizeEvent()

This method updates histograms and stats. It will call `unsuspedAllHltpu()` when the backpressure is released. It deletes the event object and calls `tryFinalizeStop()` to update the stopping satus if required.

tryFinalizeStop()

If the Processor is **Stopping** and we just deleted that last processed even, we change he state of the Processor into the **Stopped** state.

accept()

This method performs some checking on the event and calls **forceAccept()** if a problem is detected. It then checks if robs are missing and it should fetch all robs in witch case it calls **fetchAllRobs()**, or only a suaset of robs need to be fetched in which case it calls **fetchRobs()**. If no robs need to be fetched, it calls in sequence the **finalizeAccept()** and **setStateIdle()** methods.