Hlt2 line development in the new template.

For HLT2 lines in RunII, a new template has been prepared to which all existing and new HLT2 lines must be adapted. This twiki explains how to do that.

Two sets of lines have been adapted and can be used as inspiration or examples: DiMuon and LambdaC

The old lines have been removed from the package as of Moore v23r5p1. They can then still be retrieved from SVN for later conversion.

Development guidelines

- If you are interested in an introduction, how to run basic Moore scripts, have a look at the StarterKit.
- It is recommended to use the nightlies for your development, which represents the most up to date version of the software. The lhcb-head slot is most suitable.
- Use the script $MOOREROOT/tests/options/Moore_pp_2017.py for 2017 development.
- In case the nightlies are not working for a longer period, try the latest Moore version.
- Read how to commit to LHCB software Git4LHCb.
- Once committed check that you did not break the nightly tests (https://lhcb-nightlies.cern.ch/nightly/). The Moore field under lhcb-head should be green. If not check that the test Physics->2015_25ns did not produce errors connected to your development.

Hlt/Hlt2Lines

The way lines are stored in the Hlt2Lines /python/Hlt2Lines directory has changed. For every person who is responsible for a set of lines, a folder should be created. There would typically be one per working-group liason, but more can be created if this is deemed useful, please check this with your WG liason first and then post a mail to lhcb-trigger-software@cernNOSPAMPLEASE.ch.

Remember to add or update your lines in the Hlt/HltSettings directory, otherwise your changes will not work correctly.

Lines.py

This file should contain the configurable for your set of lines, which must inherit from Hlt2LinesConfigurableUser, contains the dictionary of cuts (as before) and instantiates Hlt2Line (s) in __apply_configuration__ using stages defined in Stages.py.

Please note that any imports from Stages.py must be done inside __apply_configuration__, if Stages.py contains any instantiations.

The configurable MUST be called "DirectoryName+'Lines'", where DirectoryName is the name of the directory for your lines inside Hlt2Lines.

Cuts

The cuts should be part of the __slots__ attribute, as before, but an additional level of nested dictionaries is now required, which results in the slots looking like:

{"nickname" : {"cut" : value} }

The first key is a nickname for each line you want to build, and the second is the name of the cut.
The nickname "Common" is a special case, and the cuts defined in the dictionary that is its corresponding value are added to all others when the lines are built. This allows shared cuts to be put here. It should, however, not be used as a dumping ground for all cuts.

An important requirement is that units must now be included here, as appropriate. They should no longer be part of the definition of the functor.

This is an example of part of the __slots__ attribute of the dimuon lines in the new template:

```python
{'Common' :        {'TrChi2'     :   10,
 'TrChi2Tight' :    5),
'DiMuon' :        {'MinMass'    :  2700 * MeV,
 'Pt'         :  1000 * MeV,
 'MuPt'       :   500 * MeV,
 'VertexChi2' :    25),
'JPsi' :          {'MassWindow' :    70 * MeV,
 'Pt'         :  1000 * MeV,
 'MuPt'       :   500 * MeV,
 'VertexChi2' :    25),
'JPsiHighPT' :    {'Pt'         :  4000 * MeV,
 'MassWindow' :   150 * MeV,
 'MuPt'       :   500 * MeV,
 'VertexChi2' :    25),
'B' :             {'MinMass'    :  5200 * MeV,
 'VertexChi2' :    25),
'Detached' :      {'IPChi2'     :     9,
 'DLS'        :     5}
}
```

This dictionary will now be referred to the cuts dictionary.

### Instantiating Hlt2Line (s)

Instantiation of Hlt2Line objects, which creates the Hlt2 line, must happen in the __apply_configuration__ method. The required stages are imported from Stages.py, a dictionary of {'nickname' : [stage1, stage2]} is created and this is to the "algorithms" member function of self to obtain the list algorithms that can be passed to the Hlt2Line.

This is an example __apply_configuration__ from the DiMuon Lines.py:

```python
def __apply_configuration__(self) :
    from Stages import (DiMuonFilter, JpsiFilter, Psi2SFilter,
                       BFfilter, DetachedDiMuonFilter)
    stages = {'DiMuon' : [DiMuonFilter('DiMuon')],
              'JPsi' : [JpsiFilter('JPsi')],
              'JPsiHighPT' : [JpsiFilter('JPsiHighPT')],
              'Psi2S' : [Psi2SFilter('Psi2S')],
              'Psi2SHighPT' : [Psi2SFilter('Psi2SHighPT')],
              'B' : [BFfilter('B')],
              'Detached' : [DetachedDiMuonFilter('Detached')]
    
    from HltLine.HltLine import Hlt2Line
    for (nickname, algos) in self.algorithms(stages):
        linename = 'DiMuon' + nickname if nickname != 'DiMuon' else nickname
        Hlt2Line(linename, prescale = self.prescale,
                 algos = algos, postscale = self.postscale)
```

### Using extra algorithms

Extra algorithms are run only after your line fired and do not affect the line decision. They are only relevant for Turbo lines, allowing the line to declare additional objects that should be persisted along with the
Extra algorithms are declared by defining an `extraAlgos` method on the the class that inherits from `Hlt2LinesConfigurableUser`. The method should return a dictionary of line names to dictionaries. The nested dictionaries map names to lists of stages.

```python
class MyLines(Hlt2LinesConfigurableUser):
    # …
    def extraAlgos(self):
        from Stages import ExtraPionCombiner, ExtraKaonCombiner

        extra_line_algorithms = {
            'MyLineTurbo': {
                'ExtraPions': [ExtraPionCombiner('ExtraPions')],
                'ExtraKaons': [ExtraKaonCombiner('ExtraKaons')]
            },
            'MyOtherLineTurbo': {
                # ...
            }
        }

    # …
```

Lines that do not define extra algorithms can be omitted from the dictionary.

The above example defines two sets of extra algorithms for the `MyLineNameTurbo` HLT2 line, named `ExtraPions` and `ExtraKaons`. These names are used when placing the output of the extra algorithms in the TES, such that in this example the following locations will be available for analysis offline:

```
/Event/Turbo/Hlt2MyLineTurbo/Particles
/Event/Turbo/Hlt2MyLineTurbo/ExtraPions/Particles
/Event/Turbo/Hlt2MyLineTurbo/ExtraKaons/Particles
```

Once the `extraAlgos` method is defined, the call to `self.algorithms` in `__apply_configuration__` then includes the extra algorithms in the return value. These should be passed as the `extraAlgos` keyword argument to the `Hlt2Line` constructor. Reusing the example above from the DiMuon Lines.py:

```python
def extraAlgos(self):
    # …

def __apply_configuration__(self):
    # Omitted here: import Stages and define the stages dictionary
    for (nickname, algos, extraAlgos) in self.algorithms(stages):
        linename = 'DiMuon' + nickname if nickname != 'DiMuon' else nickname
        Hlt2Line(linename, prescale = self.prescale,
                 algos = algos, postscale = self.postscale,
                 extraAlgos = extraAlgos)
```

For lines not present in the return value of the `extraAlgos` method, the call to `self.algorithms` returns an empty dictionary for those lines.

Note that any cut values defined by the extra algorithm stages must be declared in the `__slots__` dictionary of the `Hlt2LinesConfigurableUser` subclass, as for any other stage.

An example of a common extra algorithm use case is presented below.
Multiple Files with lines

If there are so many lines managed in a folder that it becomes cumbersome to have all of their cuts and instantiate them in Lines.py, multiple files may be used. Name them clearly.

Inside these files, define a function that returns the dictionary of cuts for the lines instantiated inside and use this to construct the dictionary put in the __slots__ class variable in Lines.py. Also define a function that instantiates the Hlt2Line(s) and call it from Lines.py. Make sure that the "Common" set of cuts is available everywhere, as needed.

A class can of course also be used. It is suggested to keep things minimal and as readable and maintainable as possible.

Reco.py

The RunII Hlt2 reconstruction has PT and P cuts at the same (low) value as offline, includes RICH reconstruction with proton, pion, kaon and below threshold hypothesis and muon ID. Downstream tracking is also performed up-front.

If a line requires additional reconstruction, the algorithms that perform this, should be configured and instantiated in the Reco.py file. No examples have been created yet. Functions on classes in this file should return bindMembers or Hlt2Member instances when called from Stages.py.

Inputs.py

This file should contain all imports of particles from Hlt2SharedParticles needed for the stages defined in Stages.py.

In Hlt/Hlt2SharedParticles/python/Hlt2SharedParticles you find a selection of all basic particles and also some composite particles like Ks, Lambda or DiMuons. Have a look there when you start writing your trigger line.

Stages.py

The selection steps performed by all HLT2 lines created in this folder are defined in this file; it will generally contain most of the code.

A set of classes should be defined here, which inherit from provided utility classes, and create the required selection steps. Utility classes are provided for: VoidFilter, CombineParticles /NMBodyDecays, FilterDesktop and TisTos tagging.

The definition of the stage always happens in the __init__ method, after which the base class __init__ method is called.

Constructor arguments can be chosen as appropriate to maximise reuse of code. The base class constructor accepts keyword arguments, which are passed directly to the Hlt2Member that is created in the end. This allows specific arguments to be passed to e.g. CombineParticles or FilterDesktop.

A few examples of definition of selection stages follow, and the TisTos tagging is described afterwards.

All instances of stages can be cloned to create a new instance with different properties, all base class constructor arguments can be "overwritten", and keyword arguments can be specified, which will again be passed to the underlying class.
Names and Nicknames

In short, names are the names of algorithms and nicknames refer to the cuts dictionary. For convenience, they are the same by default. In that case, the name of the algorithm is constructed as:

"Hlt2" + linename + nickname + suffix,

where suffix is usually "Filter" or "Combiner".

If this results in names that are not readable or hard to understand, more control over the name of the algorithm is possible by explicitly specifying the "nickname = some_nick" constructor argument to the different available base classes. The specified nickname will then be used to obtain the cuts from the cuts dictionary and the name of the algorithm will be constructed as:

"Hlt2" + linename + name + suffix,

This is especially useful for shared stages.

Global Event Cut or Other

Classes defining a global event cut stage or any other filter implemented using VoidFilter should inherit from Hlt2Lines.Utilities.Hlt2Filter.Hlt2VoidFilter.

from Hlt2Lines.Utilities.Hlt2Filter import Hlt2VoidFilter

# The GEC
class TrackGEC(Hlt2VoidFilter):
    def __init__(self, name):
        from Configurables import LoKi__Hybrid__CoreFactory as Factory
        modules = Factory().Modules
        for i in ['LoKiTrigger.decorators']:
            if i not in modules: modules.append(i)

        from HltTracking.Hlt2TrackingConfigurations import Hlt2BiKalmanFittedForwardTracking as Hlt2LongTracking
        tracks = Hlt2LongTracking().hlt2PrepareTracks()
        code = ("CONTAINS('%s')") % tracks.outputSelection() + " < %(NTRACK_MAX)s"
        Hlt2VoidFilter.__init__(self, 'TrackGEC', code, [tracks])

Combiner

Both CombineParticles and NMBodyDecays are supported by the Hlt2Combiner utility class, which is defined in Hlt2Lines.Utilities.Hlt2Combiner. A combiner requires several pieces of information to be created:

- Inputs
- Dependencies, i.e. selection steps that must run before the combiner, but do not produce output that should serve as input to this combiner. Only direct dependencies need to be specified.
- Decay descriptor.
- Cuts on daughters, combination and/or mother.
- Optional additional parameters of the CombineParticles /NMBodyDecays, such as a Preambulo.
- TisTos requirements. They can directly be a regex or decision name, or a key in the appropriate cuts dictionary.

In addition to the items listed above, the instantiation and cloning of a combiner are shown in the following example from the LambdaC directory in Hlt2Lines.

# The class that creates the Hlt2Combiner
from Hlt2Lines.Utilities.Hlt2Combiner import Hlt2Combiner

class LambdaC2KPPiCombiner(Hlt2Combiner):
    def __init__(self, name):
        piK_cut = "((TRCHI2DOF < %(Trk_TRCHI2DOF_MAX)s) \
        & (PT > %(Trk_PT_MIN)s) \
        & (MIPCHI2DV(PRIMARY) > %(Trk_MIPCHI2DV_MIN)s))"

        p_cut = "((TRCHI2DOF < %(Trk_TRCHI2DOF_MAX)s) \
        & (PT > %(P_PT_MIN)s) \
        & (P > %(P_P_MIN)s) \
        & (MIPCHI2DV(PRIMARY) > %(Trk_MIPCHI2DOF_MIN)s))"

        dc = {'pi+': piK_cut,
              'K+': piK_cut,
              'p+': p_cut}

        cc = "((in_range( %(Lc_AM_MIN)s, AM, %(Lc_AM_MAX)s )) + \
              & ((APT1+APT2+APT3) > %(Lc_ASUMPT_MIN)s) + \
              & (AHASCHILD((MIPCHI2DV(PRIMARY)) > %(Trk_1of3_MIPCHI2DV_MIN)s)))"

        mc = "((VFASPF(VCHI2PDOF) < %(Lc_VCHI2PDOF_MAX)s) + \
              & (BPVDIRA > %(Lc_BPVDIRA_MIN)s) + \
              & (BPVVDR < %(Lc_BPVVDR_MAX)s) + \
              & (BPVVDCHI2 > %(Lc_BPVVDCHI2_MIN)s) + \
              & (BPVLTIME() < %(Lc_BPVLTIME_MAX)s))"

from Hlt2Lines.HltPVs import PV3D
from Inputs import Hlt2LoosePions, Hlt2LooseKaons, Hlt2LooseProtons

inputs = [Hlt2LooseKaons, Hlt2LooseProtons, Hlt2LoosePions]

Hlt2Combiner.__init__(self, 'KPPi', '[Lambda_c+ -> K- p+ pi+]cc', inputs,
                      dependencies = [TrackGEC('TrackGEC'), PV3D('Hlt2')],
                      tistos = 'TisTosSpec', DaughtersCuts = dc, CombinationCut = cc,
                      MotherCut = mc, Preambulo = [])

# Instantiate the Combiners
from Inputs import Hlt2LoosePions, Hlt2LooseKaons, Hlt2LooseProtons
LambdaC2KPPi = LambdaC2KPPiCombiner('KPPi')
LambdaC2KPK = LambdaC2KPPi.clone('KPK', decay = '[Lambda_c+ -> K- p+ K+]cc',
                                  inputs = [Hlt2LooseKaons, Hlt2LooseProtons])

To create a FilterDesktop stage, the Hlt2ParticleFilter utility class should be used, which is defined in Hlt2LinesUtilities.Hlt2Filter.

Creating one is very similar to creating a combiner, as shown in the following example.

# Mass filter
from Hlt2Lines.Utilities.Hlt2Filter import Hlt2ParticleFilter

class LambdaCMassFilter(Hlt2ParticleFilter):
    def __init__(self, name, inputs):
        cut = "in_range( %(SigMass_M_MIN)s, M, %(SigMass_M_MAX)s )"
        Hlt2ParticleFilter.__init__(self, name, cut, inputs)

# Final selections
LC2KPPi = LambdaCMassFilter("KPPi", inputs = [LambdaC2KPPi])
LC2KPK = LambdaCMassFilter("KPK", inputs = [LambdaC2KPK])

**Extra particle combiner**

When using extra algorithms on Turbo lines, a common use case is to add particles that pass some combination requirements that include the line candidate itself. In these cases, one wants the extra algorithms to output only the additional particles, not the placeholder parent.

To do this, the Hlt2ExtraCombiner can be subclassed as a stage.
from Inputs import MyPions

class ExtraPionCombiner(Hlt2ExtraCombiner):
    def __init__(self, name, **kwargs):
        dc = {'pi+': 'PIDK < %(EXTRA_PION_PIDK_MAX)s'}
        cc = 'in_range(%(Q_AM_MIN)s, (AM - AM1 - AM2), %(Q_AM_MAX)s)'
        mc = '(VFASPF(VCHI2PDOF) < %(TagVCHI2PDOF_MAX)s)'
        Hlt2ExtraCombiner.__init__(self, name, **kwargs)
        # This decay descriptor is passed to CombineParticles,
        # and so should be of the 'simple' type
        '[D*(2010)+ -> D0 pi+]cc',
        DaughtersCuts=dc,
        CombinationCut=cc,
        MotherCut=mc,
        # This decay descriptor is passed to FilterDecays,
        # and so should be of the LoKi type (like for DecayTreeTuples)
        ParticlesToSave='[D*(2010)+ -> D0 ^pi+]CC',

This example uses a presumed pre-existing stage, MyD0Stage, which outputs \(D^0\) candidates, and a pion input container defined in Inputs.py, MyPions, to build a placeholder \(D^+\) candidate. The properties of the \(D^0\) are used to decide which pions should be kept. The ParticlesToSave argument of Hlt2ExtraCombiner marks which particles in the combination should be saved (remember: the Turbo line candidate, the \(D^0\) in this case, is always saved).

A more complex example may define a generic extra combiner stage which is suitable for adding extra particles of arbitrary ID, or an extra combiner where the extra particle is a composite such as a resonance or a jet. In all cases, the ParticlesToSave property would mark only the additional particles which are siblings to the Turbo line candidate in the decay tree.

Further examples are provided in the Recotest module in the Hlt2Lines package, specifically the Stages.py and Lines.py.

### Shared Stages

In case a stage should be shared between lines, either lines in the same configuration or elsewhere, the stage that your stages inherit from should have the `shared = True` argument. Please take care to give shared stages a reasonable name.

This is an example of a shared filter from the DiMuon lines.

class JpsiFilter(Hlt2ParticleFilter):
    def __init__(self, name):
        code = ('(ADMASS("J/psi(1S)") < %MassWindow)s " +
        " & (PT > %Pt)s " +
        " & (MAXTREE('mu-' == ABSID,TRCHI2DOF) < %(TrChi2Tight)s)" +
        " & (MINTREE('mu-' == ABSID,PT) > %(MuPt)s) " +
        " & (VFASPF(VCHI2PDOF) < %VertexChi2)s *)
        inputs = [TrackFittedDiMuon]
        Hlt2ParticleFilter.__init__(self, name, code, inputs, shared = True)

There are two ways of specifying a requirement on TisTos with respect to Hlt1 lines,
• The tistos constructor argument of Hlt2Combiner and Hlt2ParticleFilter
• Adding an Hlt2TisTosParticleTagger.

Both ways allow you to specify a list of cut names or TisTos specs. The TisTos specs can be copied directly from current TisTosParticleTagger instances. The cut name must be part of the cuts dictionary and resolve to a TisTos spec.

Hlt2TisTosParticleTagger can be imported from Hlt2Lines.Utilities.Hlt2TisTosFilter.

**Related Info**

To include related info to be calculated in your HLT2 line, there are several two main requirements: * Your line must be a Turbo line (if not, the variables can be calculated offline), * Timing must be acceptable

To add related info to your turbo HLT2 line, the following should be done (running example available in CharmHad/D2HHHLines.py).

A function names "relatedInfo" should be added to the Lines configurable in Lines.py. This is an example:

```python
def relatedInfo(self):
    from Hlt2Lines.Utilities.Hlt2RelatedInfo import Hlt2RelInfoConeVariables
    from Hlt2Lines.Utilities.Hlt2RelatedInfo import Hlt2RelInfoVertexIsolation
    # Cone variables
    DconeVar05 = Hlt2RelInfoConeVariables('DConeVar05')
    DconeVar10 = Hlt2RelInfoConeVariables('DConeVar10')
    # Vertex Isolation
    VertexIso = Hlt2RelInfoVertexIsolation("VertexIsolation")
    # Link info to lines
    return {'DspToKmKpPipTurbo' : [DconeVar05, DconeVar10, VertexIso]}
```

The configuration of the RelatedInfo tools is performed in the same way stages are configured, i.e. by adding entries to the cuts dictionary.

```python
'DConeVar05' : { "ConeAngle" : 0.5,
                 "Variables" : ['CONEANGLE', 'CONEMULT', 'CONEPTASYM'],
                 "Location" : 'DConeVar05' },
'DConeVar10' : { "ConeAngle" : 1.0,
                 "Variables" : ['CONEANGLE', 'CONEMULT', 'CONEPTASYM'],
                 "DaughterLocations" : {"[D_s+ -> ^K- K+ pi+]CC" : "KmConeVar10"} },
'VertexIsolation' : { "Location" : "VertexIsoInfo" }
```

Once the relatedInfo function exists, the algorithms function will return a tuple of (nickname, stages, relatedInfo), which can be passed to the Hlt2Line constructor to add the info to the line:

```python
from HltLine.HltLine import Hlt2Line
for nickname, algos, relatedInfo in self.algorithms( _stages ):
    Hlt2Line('FolderName' + nickname, prescale = self.prescale,
             algos = algos, postscale = self.postscale, Turbo = True,
             RelatedInfo = relatedInfo)
```

Currently only ConeVariables and VertexIsolation are supported, for support of more tools, please comment on https://its.cern.ch/jira/browse/LHCBPS-1445.
Script for Running Moore with Additional HLT2 lines

Use the script $MOOREROOT/tests/options/Moore_pp_2017.py on the current head of the nightlies.

(eg, lb-run --nightly lhcb-head Moore HEAD gaudirun.py $MOOREROOT/tests/options/Moore_pp_2017.py)

If you want to add additional lines, copy the script to a working directory and add the following lines.

```python
## Add your new HLT2 lines here to run them
from Configurables import HltConf
HltConf().AdditionalHlt2Lines = (["Hlt2NewLambdaC2" + n for n in ("KPPi", "KPK", "PiPPi", "PiPK")])
```

Monitoring

All trigger lines are encouraged to include monitoring of interesting quantities such as mass, transverse momentum or decay times. These histograms can be filled from any Combiner or Filter stage by passing a monitoring functor string to one of the monitoring properties of the underlying algorithm.

Preambulo

To ensure the histograms are coupled to the algorithm that created them, histograms should be defined in the Preambulo of the algorithm that created them:

```python
preambulo += ["massHisto = Gaudi.Histo1DDef('Jpsi_mass', 2900 * MeV, 3300 * MeV, 100)",
"ptHisto = Gaudi.Histo1DDef('Jpsi_PT', 0, 20 * GeV, 100)"
]
```

Functor syntax

All monitoring functors are passed a vector of particles as an argument and therefore have a specific syntax.

- to monitor a quantity returned by a functor FUN, it should be wrapped in both the monitor and the process adaptors.
- pass FUN as the first argument to "monitor", the histogram definition as the second and the name of the histogram as the third
- to monitor multiple quantities, chain them using the ">>" operator
- end the functor with "~EMPTY"

```python
motherMonitor = ("process(monitor(M, massHisto, 'Jpsi_mass'))"
" >> process(monitor(PT, ptHisto, 'Jpsi_PT'))"
" >> ~EMPTY")
```

Cuts can also be inserted in this functor to select which particles should be monitored, for example to get an idea of efficiencies.

```python
motherMonitor = ("process(monitor(M, massHisto, 'Jpsi_mass'))"
" >> select(CHILD(PIDmu, 'J/psi(1S) -> ^mu+ mu-') > 2)"
" >> process(monitor(M, massPIDHisto, 'Jpsi_mass_PID'))"
" >> ~EMPTY")
```

If multiple sets of cuts should be monitored independently, the "tee" adaptor can be used (to insert a T section in the streamer) to keep the different parts independent.

```python
monitor = ("process(monitor(Dst_D0_DMASS, probeDMass, 'Dst_D0_DMAss'))"
" >> tee(select(CHILD(DLLK, ABSID == 'K+') > 5)"
"   >> monitor(Dst_D0_DMAss, effKDMass, 'Dst_D0_effKDMass'))"
" >> tee(select(CHILD(DLLK, ABSID == 'pi+') > 5)"
```
Note that in the case you want to monitor the decay products, the corresponding monitor should be a dictionary of IDs and monitors, similarly to how you set DaughtersCuts in Combine Particles, e.g.

daughterMonitor = { "mu+" : { "process(monitor(ETA, etaHist_mu_muPi, 'mu_muPi_ETA'))" >> ~EMPTY" }}

Combiners

Combiner stages can monitor after daughter, combination and mother cuts; after cut(s) have been applied. The properties are called "DaughtersMonitors", "CombinationMonitor" and "MotherMonitor".

class SomeCombiner(Hlt2Combiner):
    def __init__(self, name):
        mc = ...
        preambulo = ["massHisto = Gaudi.Histo1DDef('Lambdac_mass', 2180 * MeV, 2380 * MeV, 100)"
                     motherMonitor = ("process(monitor(M, massHisto, 'Lambdac_mass'))"
                                     >> ~EMPTY")
        Hlt2Combiner.__init__(self, 'Combiner', "A -> B C", inputs = [...],
                               dependencies = [...], Preambulo = preambulo,
                               MotherMonitor = motherMonitor)

Filters

Filter stages can monitor input or output particles using the "PreMonitor" and "PostMonitor" properties

class JpsiFilter(Hlt2ParticleFilter):
    def __init__(self, name):
        code = "...
        preambulo = ["massHisto = Gaudi.Histo1DDef('Jpsi_mass', 3100 * MeV, 3300 * MeV, 100)"
                     monitor = "process(monitor(M, massHisto, 'Jpsi_mass'))"
                       >> ~EMPTY")
        Hlt2ParticleFilter.__init__(self, name, code, inputs = [...], shared = True,
                                   Preambulo = preambulo,
                                   PostMonitor = monitor)

Default Monitoring Histograms

During configuration, default mass monitoring histograms will be added, based on the decay descriptor of the last combiner in the line. If cuts on the combination mass or mass are applied in the last combiner or any subsequent filters, the code will attempt to parse these to determine the tightest cuts applied to define the edges of the histogram.

If any monitoring plots have been added to a combiner or filter that are part of the line, no default mass histogram will be added.

If a line is intended for a blind analysis and the default monitoring histogram should not be created, please indicate this in the BlindHlt2Lines page.

Testing Monitoring Histograms

To test monitoring histograms, add the following options to the script described above and check the resulting ROOT file:

from Configurables import HistogramPersistencySvc, RootHistCnv__PersSvc
from Configurables import HltMonitoringConf
To test the monitoring functors without running all of Moore, add the following lines in addition to the ones mentioned above:

```python
from GaudiPython.Bindings import AppMgr
from LoKiCore.decorators import *
```

You can then run the script with "python -i" and test things interactively

```python
>>> testHisto = Gaudi.Histo1DDef('mass', 1000, 2000, 100)
>>> testFun = process(monitor(M, testHisto, 'mass')) >> ~EMPTY
```

If you get an error message that the histogram could not be booked, the syntax of the functor is correct. The error arises because the functor is being built outside an algorithm. When printing the functor, You should see something like the following:

```python
>>> print testFun
(process(M)>> (~EMPTY)
```

### Testing the rate of your new Hlt2 line

To test the rate of your new line use the scripts given here.

### Update the settings directory

The line properties need to be added and or updated in Hlt/HltSettings as the settings overwrite the properties specified in Hlt2Lines. Every Hlt2Lines subfolder has a corresponding subfolder in Hlt/HltSettings/python/HltSettings. If you created, a new subfolder in Hlt2Lines, create a new subfolder in Hlt/HltSettings as well. Always update the Draft2016 file to make sure that your changes get picked up in the nightly tests and the next TCK. If you added a new subfolder, make sure it is added to the main settings file. Otherwise your lines will not be picked up.

### Questions and problems

In case of questions or problems, please email lhcb-hlt-software@cernNOSPAMPLEASE.ch and/or come to the HltOps meeting on Fridays at 15:00.

### FAQ

I am using the nightly version and I see an error like

```
# setting LC_ALL to "C"
# --> Including file MooreOptions.py'
# WARNING: : Module HltConf.Configuration not found (needed for configurable HltConf)
```

This most likely means that this nightly version is broken. Go to (https://lhcb-nightlies.cern.ch/nightly/) and check if the Moore field under lhcb-head is read. Then see if the Physics->2015_25ns produced the same error message.

-- RoelAaij - 2017-04-04