DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering Reference Manual

0.1

Generated by Doxygen 1.3.2

Tue May 4 15:32:38 2004

Contents

Chapter 1

DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering Data Structure Index

1.1 DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering Data Structures

Here are the data structures with brief descriptions:

Chapter 2

DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering File Index

2.1 DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering File List

Here is a list of all files with brief descriptions:

Chapter 3

DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering Data Structure Documentation

3.1 avl_node Struct Reference

```
#include <avl.h>
```

Collaboration diagram for avl_node:

avl_node
avl_link

Data Fields

- avl_node ∗ avl_link [2]
- void ∗ avl_data
- signed char avl_balance

3.1.1 Field Documentation

3.1.1.1 signed char avl_node::avl_balance

Definition at line 73 of file avl.h.

Referenced by avl_copy(), avl_delete(), and avl_probe().

3.1.1.2 void∗ avl_node::avl_data

Definition at line 72 of file avl.h.

Referenced by avl_copy(), avl_delete(), avl_destroy(), avl_find(), avl_probe(), avl_t_copy(), avl_t_cur(), avl_t_find(), avl_t_first(), avl_t_last(), avl_t_next(), avl_t_prev(), and avl_t_replace().

3.1.1.3 struct avl_node* avl_node::avl_link[2]

Definition at line 71 of file avl.h.

Referenced by avl_copy(), avl_delete(), avl_destroy(), avl_find(), avl_probe(), avl_t_find(), avl_t_first(), avl_-t_last(), avl_t_next(), and avl_t_prev().

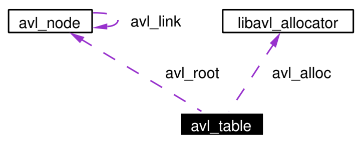
The documentation for this struct was generated from the following file:

- avl.h

3.2 avl_table Struct Reference

```
#include <avl.h>
```

Collaboration diagram for avl_table:



Data Fields

- avl_node ∗ avl_root
- avl_comparison_func ∗ avl_compare
- void ∗ avl_param
- libavl_allocator ∗ avl_alloc
- size_t avl_count
- unsigned long avl_generation

3.2.1 Field Documentation

3.2.1.1 struct libavl_allocator∗ avl_table::avl_alloc

Definition at line 63 of file avl.h.

Referenced by avl_copy(), avl_create(), and avl_destroy().

3.2.1.2 avl_comparison_func∗ avl_table::avl_compare

Definition at line 61 of file avl.h.

Referenced by avl_copy(), avl_create(), avl_find(), and avl_t_find().

3.2.1.3 size_t avl_table::avl_count

Definition at line 64 of file avl.h.

Referenced by avl_copy(), and avl_create().

3.2.1.4 unsigned long avl_table::avl_generation

Definition at line 65 of file avl.h.

Referenced by avl_create(), avl_t_copy(), avl_t_find(), avl_t_first(), avl_t_init(), avl_t_insert(), avl_t_last(), avl_t_next(), and avl_t_prev().

3.2.1.5 void∗ avl_table::avl_param

Definition at line 62 of file avl.h.

Referenced by avl_copy(), avl_create(), avl_destroy(), avl_find(), and avl_t_find().

3.2.1.6 struct avl_node∗ avl_table::avl_root

Definition at line 60 of file avl.h.

Referenced by avl_create(), avl_destroy(), avl_find(), avl_t_find(), avl_t_first(), and avl_t_last().

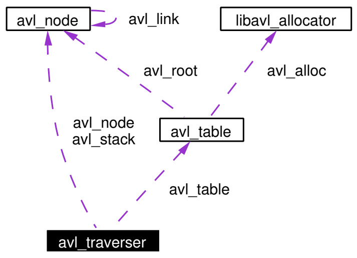
The documentation for this struct was generated from the following file:

- avl.h

3.3 avl_traverser Struct Reference

`#include <avl.h>`

Collaboration diagram for avl_traverser:



Data Fields

- avl_table ∗ avl_table
- avl_node ∗ avl_node
- avl_node ∗ avl_stack [AVL_MAX_HEIGHT]
- size_t avl_height
- unsigned long avl_generation

3.3.1 Field Documentation

3.3.1.1 unsigned long avl_traverser::avl_generation

Definition at line 84 of file avl.h.

Referenced by avl_t_copy(), avl_t_find(), avl_t_first(), avl_t_init(), avl_t_insert(), avl_t_last(), avl_t_next(), and avl_t_prev().

3.3.1.2 size_t avl_traverser::avl_height

Definition at line 83 of file avl.h.

Referenced by avl_t_copy(), avl_t_find(), avl_t_first(), avl_t_init(), avl_t_last(), avl_t_next(), and avl_t_prev().

3.3.1.3 struct avl_node∗ avl_traverser::avl_node

Definition at line 80 of file avl.h.

Referenced by avl_t_copy(), avl_t_cur(), avl_t_find(), avl_t_first(), avl_t_init(), avl_t_insert(), avl_t_last(), avl_t_next(), avl_t_prev(), and avl_t_replace().

3.3.1.4 struct avl_node∗ avl_traverser::avl_stack[AVL_MAX_HEIGHT]

Definition at line 81 of file avl.h.

Referenced by avl_t_copy(), avl_t_find(), avl_t_first(), avl_t_last(), avl_t_next(), and avl_t_prev().

3.3.1.5 struct avl_table* avl_traverser::avl_table

Definition at line 79 of file avl.h.

Referenced by avl_t_copy(), avl_t_find(), avl_t_first(), avl_t_init(), avl_t_insert(), avl_t_last(), avl_t_next(), and avl_t_prev().

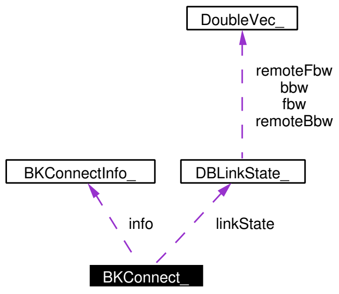
The documentation for this struct was generated from the following file:

- avl.h

3.4 BKConnect_ Struct Reference

`#include <primaryPath_util.h>`

Collaboration diagram for BKConnect_:



Data Fields

- long neighbId
- DBLinkState ∗ linkState
- BKConnectInfo info

3.4.1 Field Documentation

3.4.1.1 BKConnectInfo BKConnect_::info

Definition at line 18 of file primaryPath_util.h.

Referenced by fillTopo(), and makeScore().

3.4.1.2 DBLinkState∗ BKConnect_::linkState

Definition at line 17 of file primaryPath_util.h.

Referenced by bkConnectVecGet(), bkConnectVecPopBack(), bkConnectVecPushBack(), bkConnectVecSet(), fillTopo(), initScore(), makeScore(), and updateNodeInfoOnElect().

3.4.1.3 long BKConnect_::neighbId

Definition at line 16 of file primaryPath_util.h.

Referenced by bellmanKalaba(), bkConnectVecGet(), bkConnectVecPopBack(), bkConnectVecPushBack(), bkConnectVecSet(), fillTopo(), and printTopo().

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

- primaryPath_util.h

3.5 BKConnectInfo_ Struct Reference

```
#include <primaryPath_util.h>
```

Data Fields

- double gain [NB_OA]

3.5.1 Field Documentation

3.5.1.1 double BKConnectInfo_::gain[NB_OA]

Definition at line 11 of file primaryPath_util.h.

Referenced by fillTopo(), and makeScore().

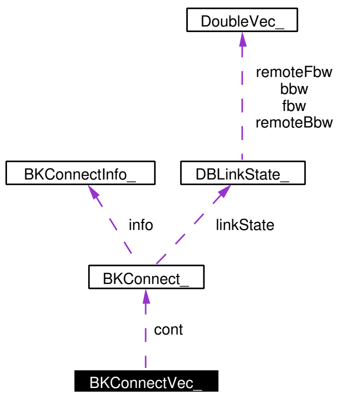
The documentation for this struct was generated from the following file:

- primaryPath_util.h

3.6 BKConnectVec_ Struct Reference

`#include <primaryPath_util.h>`

Collaboration diagram for BKConnectVec_:



Data Fields

- long size
- long top
- BKConnect ∗ cont

3.6.1 Field Documentation

3.6.1.1 BKConnect∗ BKConnectVec_::cont

Definition at line 25 of file primaryPath_util.h.

Referenced by bellmanKalaba(), bkConnectVecCopy(), bkConnectVecDestroy(), bkConnectVecEnd(), bkConnectVecGet(), bkConnectVecPopBack(), bkConnectVecPushBack(), bkConnectVecResize(), bk-ConnectVecSet(), fillTopo(), initScore(), noLoop(), printTopo(), and updateRequest().

3.6.1.2 long BKConnectVec_::size

Definition at line 23 of file primaryPath_util.h.

Referenced by bkConnectVecCopy(), bkConnectVecEnd(), bkConnectVecGet(), bkConnectVecPush-Back(), bkConnectVecResize(), and bkConnectVecSet().

3.6.1.3 long BKConnectVec_::top

Definition at line 24 of file primaryPath_util.h.

Referenced by bellmanKalaba(), bkConnectVecCopy(), bkConnectVecEnd(), bkConnectVecPopBack(), bkConnectVecPushBack(), bkConnectVecSet(), fillTopo(), initScore(), and printTopo().

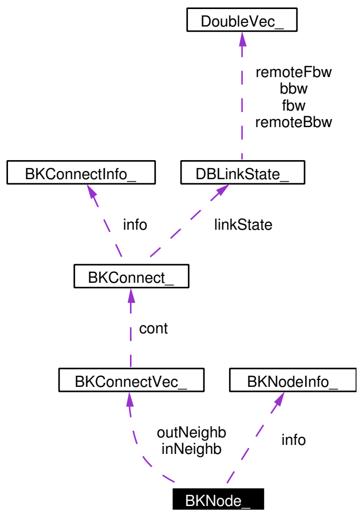
The documentation for this struct was generated from the following file:

- primaryPath_util.h

3.7 BKNode_ Struct Reference

`#include <primaryPath_util.h>`

Collaboration diagram for BKNode_:



Data Fields

- long nodeId
- BKConnectVec inNeighb
- BKConnectVec outNeighb
- long neighbInd
- BKNodeInfo info

3.7.1 Field Documentation

3.7.1.1 BKNodeInfo BKNode_::info

Definition at line 54 of file primaryPath_util.h.

Referenced by bellmanKalaba(), and noLoop().

3.7.1.2 BKConnectVec BKNode_::inNeighb

Definition at line 51 of file primaryPath_util.h.

Referenced by bellmanKalaba(), bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecPopBack(), bkNode-VecPushBack(), bkNodeVecSet(), fillTopo(), initScore(), noLoop(), printTopo(), and updateRequest().

3.7.1.3 long BKNode_::neighbInd

Definition at line 53 of file primaryPath_util.h.

Referenced by bellmanKalaba(), bkNodeVecPopBack(), bkNodeVecPushBack(), bkNodeVecSet(), fillTopo(), noLoop(), printTopo(), and updateRequest().

3.7.1.4 long BKNode_::nodeId

Definition at line 50 of file primaryPath_util.h.

Referenced by bkNodeVecPopBack(), bkNodeVecPushBack(), bkNodeVecSet(), fillTopo(), noLoop(), and printTopo().

3.7.1.5 BKConnectVec BKNode_::outNeighb

Definition at line 52 of file primaryPath_util.h.

Referenced by bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecPopBack(), bkNodeVecPushBack(), bkNodeVecSet(), fillTopo(), and printTopo().

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

- primaryPath_util.h

3.8 BKNodeInfo_ Struct Reference

```
#include <primaryPath_util.h>
```

Data Fields

- long newNeighbInd
- double newCost
- double cost
- double newSum [NB_OA]
- double sum [NB_OA]

3.8.1 Field Documentation

3.8.1.1 double BKNodeInfo_::cost

Definition at line 43 of file primaryPath_util.h.

Referenced by bellmanKalaba().

3.8.1.2 double BKNodeInfo_::newCost

Definition at line 42 of file primaryPath_util.h.

Referenced by bellmanKalaba().

3.8.1.3 long BKNodeInfo_::newNeighbInd

Definition at line 41 of file primaryPath_util.h.

Referenced by bellmanKalaba(), and noLoop().

3.8.1.4 double BKNodeInfo_::newSum[NB_OA]

Definition at line 44 of file primaryPath_util.h.

3.8.1.5 double BKNodeInfo_::sum[NB_OA]

Definition at line 45 of file primaryPath_util.h.

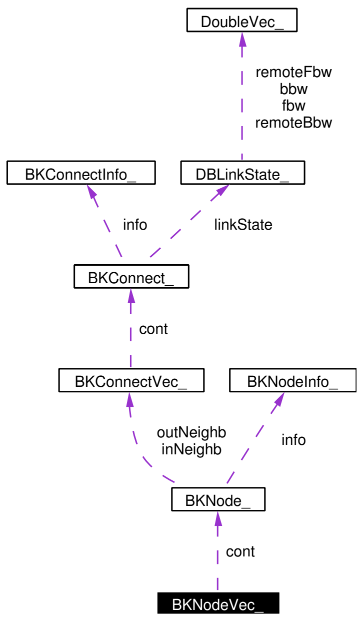
The documentation for this struct was generated from the following file:

- primaryPath_util.h

3.9 BKNodeVec_ Struct Reference

`#include <primaryPath_util.h>`

Collaboration diagram for BKNodeVec_:



Data Fields

- long size
- long top
- BKNode * cont

3.9.1 Field Documentation

3.9.1.1 BKNode* BKNodeVec_::cont

Definition at line 61 of file primaryPath_util.h.

Referenced by activateNodeInfo(), bellmanKalaba(), bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecGet(), bkNodeVecInit(), bkNodeVecNew(), bkNodeVecPopBack(), bkNodeVecPushBack(), bkNodeVecResize(), bkNodeVecSet(), fillTopo(), initScore(), makeScore(), noLoop(), and updateNodeInfoOnElect().

3.9.1.2 long BKNodeVec_::size

Definition at line 59 of file primaryPath_util.h.

Referenced by bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecGet(), bkNodeVecInit(), bkNodeVecNew(), bkNodeVecPushBack(), bkNodeVecResize(), and bkNodeVecSet().

3.9.1.3 long BKNodeVec_::top

Definition at line 60 of file primaryPath_util.h.

Referenced by bkNodeVecEnd(), bkNodeVecInit(), bkNodeVecNew(), bkNodeVecPopBack(), bkNodeVecPushBack(), bkNodeVecSet(), fillTopo(), and initScore().

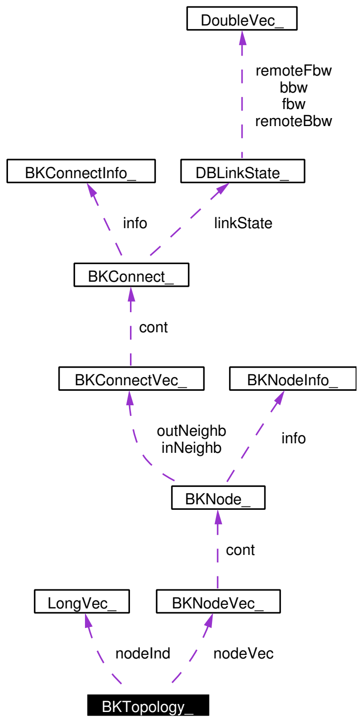
The documentation for this struct was generated from the following file:

- primaryPath_util.h

3.10 BKTopology_ Struct Reference

`#include <primaryPath_util.h>`

Collaboration diagram for BKTopology_:



Data Fields

- BKNodeVec nodeVec
- LongVec nodeInd
- long nbNodes
- long nbLinks

3.10.1 Field Documentation

3.10.1.1 long BKTopology_::nbLinks

Definition at line 79 of file primaryPath_util.h.

Referenced by fillTopo(), and makeScore().

3.10.1.2 long BKTopology_::nbNodes

Definition at line 78 of file primaryPath_util.h.

Referenced by fillTopo().

3.10.1.3 LongVec BKTopology_::nodeInd

Definition at line 77 of file primaryPath_util.h.

Referenced by activateNodeInfo(), bellmanKalaba(), endTopo(), fillTopo(), initScore(), initTopo(), makeScore(), noLoop(), printTopo(), updateNodeInfoOnElect(), and updateRequest().

3.10.1.4 BKNodeVec BKTopology_::nodeVec

Definition at line 76 of file primaryPath_util.h.

Referenced by activateNodeInfo(), bellmanKalaba(), endTopo(), fillTopo(), initScore(), initTopo(), makeScore(), noLoop(), printTopo(), updateNodeInfoOnElect(), and updateRequest().

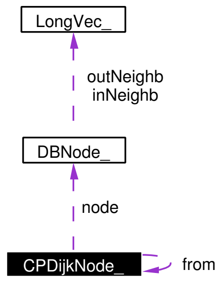
The documentation for this struct was generated from the following file:

- primaryPath_util.h

3.11 CPDijkNode_ Struct Reference

```
#include <dijkstra.h>
```

Collaboration diagram for CPDijkNode_:



Data Fields

- CPDijkNode_ * from
- DBNode * node
- double val
- bool marked

3.11.1 Field Documentation

3.11.1.1 struct CPDijkNode_* CPDijkNode_::from

Definition at line 7 of file dijkstra.h.

Referenced by computeBackup(), and computeCost().

3.11.1.2 bool CPDijkNode_::marked

Definition at line 10 of file dijkstra.h.

Referenced by computeBackup().

3.11.1.3 DBNode* CPDijkNode_::node

Definition at line 8 of file dijkstra.h.

Referenced by computeBackup(), and computeCost().

3.11.1.4 double CPDijkNode_::val

Definition at line 9 of file dijkstra.h.

Referenced by computeBackup().

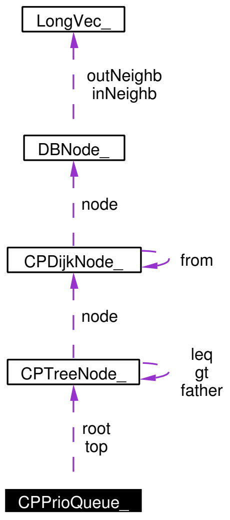
The documentation for this struct was generated from the following file:

- dijkstra.h

3.12 CPPrioQueue_ Struct Reference

`#include <dijkstra.h>`

Collaboration diagram for CPPrioQueue_:



Data Fields

- CPTreeNode ∗ root
- CPTreeNode ∗ top
- long size

3.12.1 Field Documentation

3.12.1.1 CPTreeNode∗ CPPrioQueue_::root

Definition at line 30 of file dijkstra.h.

Referenced by CPendPQ(), CPinitPQ(), CPinsertPQ(), and CPpopTop().

3.12.1.2 long CPPrioQueue_::size

Definition at line 32 of file dijkstra.h.

Referenced by CPendPQ(), CPinitPQ(), CPinsertPQ(), and CPpopTop().

3.12.1.3 CPTreeNode∗ CPPrioQueue_::top

Definition at line 31 of file dijkstra.h.

Referenced by CPendPQ(), CPinitPQ(), CPinsertPQ(), and CPpopTop().

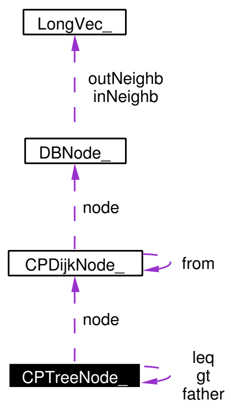
The documentation for this struct was generated from the following file:

- dijkstra.h

3.13 CPTreeNode_ Struct Reference

```
#include <dijkstra.h>
```

Collaboration diagram for CPTreeNode_:



Data Fields

- CPTreeNode_ ∗ father
- CPTreeNode_ ∗ leq
- CPTreeNode_ ∗ gt
- double key
- CPDijkNode ∗ node

3.13.1 Field Documentation

3.13.1.1 struct CPTreeNode_∗ CPTreeNode_::father

Definition at line 18 of file dijkstra.h.

Referenced by CPinsertPQ(), and CPpopTop().

3.13.1.2 struct CPTreeNode_∗ CPTreeNode_::gt

Definition at line 20 of file dijkstra.h.

Referenced by CPinsertPQ(), and CPpopTop().

3.13.1.3 double CPTreeNode_::key

Definition at line 21 of file dijkstra.h.

Referenced by CPinsertPQ().

3.13.1.4 struct CPTreeNode_∗ CPTreeNode_::leq

Definition at line 19 of file dijkstra.h.

Referenced by CPinsertPQ(), and CPpopTop().

3.13.1.5 CPDijkNode∗ CPTreeNode_::node

Definition at line 22 of file dijkstra.h.

Referenced by CPinsertPQ(), and CPpopTop().

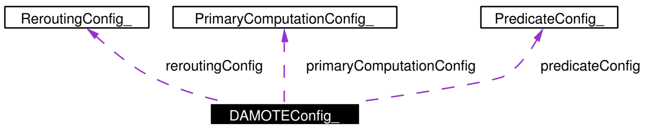
The documentation for this struct was generated from the following file:

- dijkstra.h

3.14 DAMOTEConfig_ Struct Reference

```
#include <setup.h>
```

Collaboration diagram for DAMOTEConfig_:



Data Fields

- PrimaryComputationConfig primaryComputationConfig
- PredicateConfig predicateConfig
- ReroutingConfig reroutingConfig

3.14.1 Field Documentation

3.14.1.1 PredicateConfig DAMOTEConfig_::predicateConfig

Definition at line 102 of file setup.h.

Referenced by capacityClause(), and isValidRequestLink().

3.14.1.2 PrimaryComputationConfig DAMOTEConfig_::primaryComputationConfig

Definition at line 101 of file setup.h.

Referenced by activateNodeInfo(), initScore(), makeScore(), and updateNodeInfoOnElect().

3.14.1.3 ReroutingConfig DAMOTEConfig_::reroutingConfig

Definition at line 103 of file setup.h.

Referenced by makeRerouteScore().

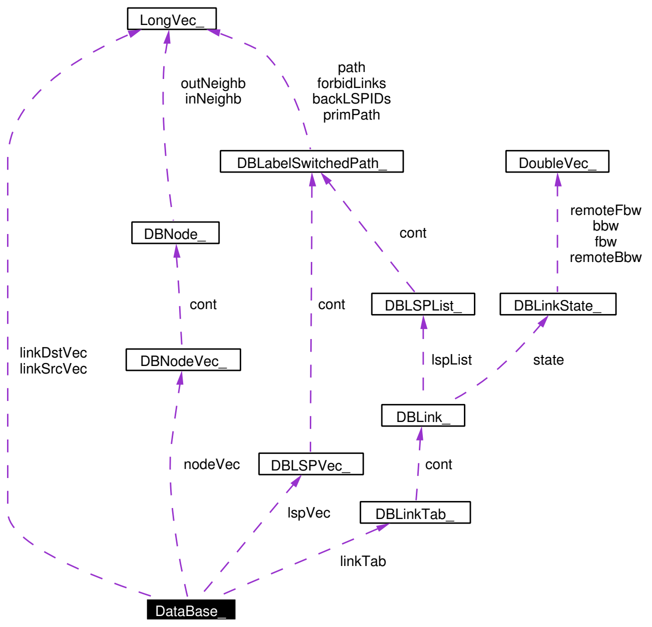
The documentation for this struct was generated from the following file:

- setup.h

3.15 DataBase_ Struct Reference

`#include <database_util.h>`

Collaboration diagram for DataBase_:



Data Fields

- long id

    ID of the node to which this database is related.

- long nbNodes
- long nbLinks
- DBNodeVec nodeVec

    Array of all nodes.

- DBLSPVec lspVec

    Array of all LSPs established.

- DBLinkTab linkTab

    Bidimentionnal array of Links. It is nodeVec ∗ nodeVec large.

- LongVec linkSrcVec
- LongVec linkDstVec

3.15.1 Field Documentation

3.15.1.1 long DataBase_::id

ID of the node to which this database is related.

Used to garantee that when this agent is used in simulator mode, no illegal information is accessed.

Definition at line 121 of file database_util.h.

Referenced by DBaddLSP(), DBgetID(), and DBnew().

3.15.1.2 LongVec DataBase_::linkDstVec

Definition at line 131 of file database_util.h.

Referenced by DBaddLink(), DBdestroy(), DBgetLinkDst(), DBnew(), and DBremoveLink().

3.15.1.3 LongVec DataBase_::linkSrcVec

Definition at line 130 of file database_util.h.

Referenced by computeBackup(), DBaddLink(), DBaddLSP(), DBdestroy(), DBgetLinkSrc(), DBnew(), and DBremoveLink().

3.15.1.4 DBLinkTab DataBase_::linkTab

Bidimentionnal array of Links. It is nodeVec $*$ nodeVec large.

Definition at line 129 of file database_util.h.

Referenced by DBaddLink(), DBaddLSP(), DBdestroy(), DBgetLinkID(), DBgetLinkLSPs(), DBgetLink-State(), DBnew(), DBprintDB(), DBremoveLink(), and DBsetLinkState().

3.15.1.5 DBLSPVec DataBase_::lspVec

Array of all LSPs established.

Definition at line 127 of file database_util.h.

Referenced by DBaddLSP(), DBdestroy(), DBgetLSP(), and DBnew().

3.15.1.6 long DataBase_::nbLinks

Definition at line 123 of file database_util.h.

Referenced by DBaddLink(), DBgetNbLinks(), DBnew(), DBremoveLink(), and DBremoveNode().

3.15.1.7 long DataBase_::nbNodes

Definition at line 122 of file database_util.h.

Referenced by DBaddNode(), DBgetNbNodes(), and DBnew().

3.15.1.8 DBNodeVec DataBase_::nodeVec

Array of all nodes.

Definition at line 125 of file database_util.h.

Referenced by computeBackup(), DBaddLink(), DBaddNode(), DBdestroy(), DBgetMaxNodeID(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), DBnew(), DBprintDB(), DBremoveLink(), and DBremoveNode().

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

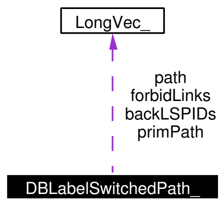
- database_util.h

3.16 DBLabelSwitchedPath_ Struct Reference

LSP structure.

`#include <database_st.h>`

Collaboration diagram for DBLabelSwitchedPath_:



Data Fields

- long id

  id of the LSP

- long noContentionId

  soft preemption, the id of the preempted LSP, of which resources can be used

- int precedence

  preemption (or priority) level

- double bw [NB_OA]

  LSP bandwidth.

- LongList forbidLinks

  list of the link colors that can't be used

- LongList path

  LSP path.

- DBLSPType type

  can be PRIM, GLOBAL_BACK or LOCAL_BACK

- long primID

  id of the primary LSP if the LSP is a backup

- LongList primPath

  path of the primary LSP if the LSP is a backup

- LongList backLSPIDs

  list of associated backup LSPs (?)

3.16.1 Detailed Description

LSP structure.

Label Switched Path representation, used by DBaddLSP. It is often needed to translate LSPRequest (used when computing) to DBLabelSwitchedPath (used when adding a LSP to the database).

Definition at line 24 of file database_st.h.

3.16.2 Field Documentation

3.16.2.1 LongList DBLabelSwitchedPath_::backLSPIDs

list of associated backup LSPs (?)

Definition at line 35 of file database_st.h.

Referenced by DBlspCopy(), DBlspDestroy(), DBlspEnd(), DBlspInit(), and DBlspNew().

3.16.2.2 double DBLabelSwitchedPath_::bw[NB_OA]

LSP bandwidth.

Note that to be Diff-Serv Aware TE compliant, this field should be different from 0 only for one OA, because multiple OAs are not allowed on the same LSP

Definition at line 29 of file database_st.h.

Referenced by chooseReroutedLSPs(), computeBackup(), computeCost(), DBlspCompare(), DBlsp-Copy(), DBlspInit(), evalLS(), isValidLSPLink(), and updateLS().

3.16.2.3 LongList DBLabelSwitchedPath_::forbidLinks

list of the link colors that can't be used

Definition at line 30 of file database_st.h.

Referenced by DBlspCopy(), DBlspDestroy(), DBlspEnd(), DBlspInit(), DBlspNew(), evalLS(), and is-ValidLSPLink().

3.16.2.4 long DBLabelSwitchedPath_::id

id of the LSP

Definition at line 26 of file database_st.h.

Referenced by chooseReroutedLSPs(), computeBackup(), computeCost(), DBaddLSP(), DBlspCompare(), DBlspCopy(), DBprintLink(), evalLS(), and isValidLSPLink().

3.16.2.5 long DBLabelSwitchedPath_::noContentionId

soft preemption, the id of the preempted LSP, of which resources can be used

Definition at line 27 of file database_st.h.

Referenced by DBaddLSP(), DBlspCopy(), DBlspInit(), DBlspNew(), evalLS(), isValidLSPLink(), and updateLS().

3.16.2.6 LongList DBLabelSwitchedPath_::path

LSP path.

Definition at line 31 of file database_st.h.

Referenced by computeBackup(), DBaddLSP(), DBlspCopy(), DBlspDestroy(), DBlspEnd(), DBlspInit(), DBlspNew(), evalLS(), isValidLSPLink(), and updateLS().

3.16.2.7 int DBLabelSwitchedPath_::precedence

preemption (or priority) level

Definition at line 28 of file database_st.h.

Referenced by chooseReroutedLSPs(), computeBackup(), computeCost(), DBaddLSP(), DBlspCompare(), DBlspCopy(), evalLS(), isValidLSPLink(), and updateLS().

3.16.2.8 long DBLabelSwitchedPath_::primID

id of the primary LSP if the LSP is a backup

Definition at line 33 of file database_st.h.

Referenced by DBlspCopy(), evalLS(), and isValidLSPLink().

3.16.2.9 LongList DBLabelSwitchedPath_::primPath

path of the primary LSP if the LSP is a backup

Definition at line 34 of file database_st.h.

Referenced by computeCost(), DBaddLSP(), DBlspCopy(), DBlspDestroy(), DBlspEnd(), DBlspInit(), DBlspNew(), evalLS(), and updateLS().

3.16.2.10 DBLSPType DBLabelSwitchedPath_::type

can be PRIM, GLOBAL_BACK or LOCAL_BACK

Definition at line 32 of file database_st.h.

Referenced by DBaddLSP(), DBlspCopy(), evalLS(), isValidLSPLink(), and updateLS().

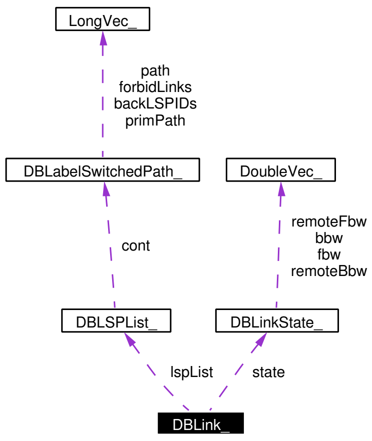
The documentation for this struct was generated from the following file:

- database_st.h

3.17 DBLink_ Struct Reference

`#include <database_util.h>`

Collaboration diagram for DBLink_:



Data Fields

- long id
- DBLinkState state
- DBLSPList lspList

  List of LSPs passing through the Link.

3.17.1 Field Documentation

3.17.1.1 long DBLink_::id

Definition at line 36 of file database_util.h.

Referenced by DBaddLink(), DBaddLSP(), and DBgetLinkID().

3.17.1.2 DBLSPList DBLink_::lspList

List of LSPs passing through the Link.

Definition at line 39 of file database_util.h.

Referenced by DBaddLSP(), DBgetLinkLSPs(), DBlinkDestroy(), DBlinkEnd(), DBlinkInit(), DBlink-New(), and DBprintLink().

3.17.1.3 DBLinkState DBLink_::state

Definition at line 37 of file database_util.h.

Referenced by DBaddLink(), DBaddLSP(), DBgetLinkState(), DBlinkDestroy(), DBlinkEnd(), DBlinkInit(), DBlinkNew(), DBprintLink(), and DBsetLinkState().

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

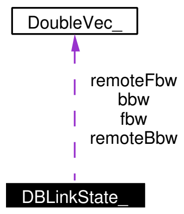
- database_util.h

3.18 DBLinkState_ Struct Reference

Link state structure.

`#include <database_st.h>`

Collaboration diagram for DBLinkState_:



Data Fields

- long color

  color

- double cap [NB_OA]

  capacity per OA

- double rbw [NB_OA][NB_PREEMPTION]

  bandwidth reserved by all LSPs (primary and backup)

- double pbw [NB_OA][NB_PREEMPTION]

  bandwidth reserved by all primary LSPs

- DoubleVec bbw [NB_OA][NB_PREEMPTION]

  bandwidth needed on this link when a failure on a certain link of the topology happens

- DoubleVec remoteBbw [NB_OA][NB_PREEMPTION]

  idem as bbw

- DoubleVec fbw [NB_OA][NB_PREEMPTION]

  bandwidth freed on this link when a failure on a certain link of the topology happens

- DoubleVec remoteFbw [NB_OA][NB_PREEMPTION]

  idem as fbw

3.18.1 Detailed Description

Link state structure.

This is the information maintained for each link.

Definition at line 55 of file database_st.h.

3.18.2 Field Documentation

3.18.2.1 DoubleVec DBLinkState_::bbw[NB_OA][NB_PREEMPTION]

bandwidth needed on this link when a failure on a certain link of the topology happens

This is probably a ”max” value.

Definition at line 65 of file database_st.h.

Referenced by DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), DBlinkStateNew(), and updateLS().

3.18.2.2 double DBLinkState_::cap[NB_OA]

capacity per OA

Definition at line 59 of file database_st.h.

Referenced by capacityClause(), computeCost(), DBlinkStateCopy(), DBprintLink(), initScore(), makeScore(), and updateNodeInfoOnElect().

3.18.2.3 long DBLinkState_::color

color

Definition at line 57 of file database_st.h.

Referenced by colorClause(), and DBlinkStateCopy().

3.18.2.4 DoubleVec DBLinkState_::fbw[NB_OA][NB_PREEMPTION]

bandwidth freed on this link when a failure on a certain link of the topology happens

This is probably a ”max” value.

Definition at line 69 of file database_st.h.

Referenced by DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), DBlinkStateNew(), and updateLS().

3.18.2.5 double DBLinkState_::pbw[NB_OA][NB_PREEMPTION]

bandwidth reserved by all primary LSPs

This is the sum of the bandwidths reserved by all primary LSPs

Definition at line 63 of file database_st.h.

Referenced by DBlinkStateCopy(), DBprintLink(), initScore(), makeScore(), and updateLS().

3.18.2.6 double DBLinkState_::rbw[NB_OA][NB_PREEMPTION]

bandwidth reserved by all LSPs (primary and backup)

This is not the sum of the reserved bandwidths because of backup bandwidth aggregation.

Definition at line 61 of file database_st.h.

Referenced by capacityClause(), chooseReroutedLSPs(), computeCost(), DBlinkStateCopy(), DBprint-Link(), makeRerouteScore(), and updateLS().

3.18.2.7 DoubleVec DBLinkState_::remoteBbw[NB_OA][NB_PREEMPTION]

idem as bbw

Structured differently and not used

Definition at line 67 of file database_st.h.

Referenced by DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), and DBlinkStateNew().

3.18.2.8 DoubleVec DBLinkState_::remoteFbw[NB_OA][NB_PREEMPTION]

idem as fbw

Structured differently and not used.

Definition at line 70 of file database_st.h.

Referenced by DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), and DBlinkStateNew().

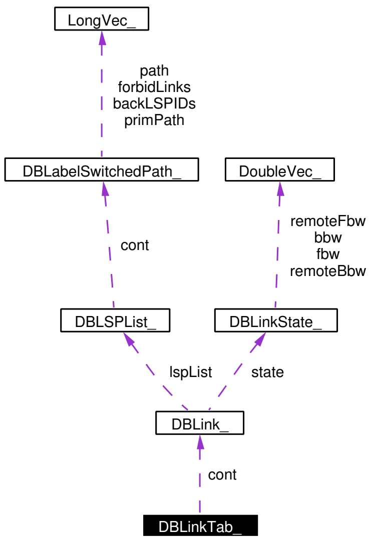
The documentation for this struct was generated from the following file:

- database_st.h

3.19 DBLinkTab_ Struct Reference

`#include <database_util.h>`

Collaboration diagram for DBLinkTab_:



Data Fields

- long size
- DBLink *** cont

3.19.1 Field Documentation

3.19.1.1 DBLink*** DBLinkTab_::cont

Definition at line 96 of file database_util.h.

Referenced by DBlinkTabDestroy(), DBlinkTabEnd(), DBlinkTabInit(), DBlinkTabNew(), DBlinkTab-Remove(), DBlinkTabResize(), DBlinkTabSet(), and DBprintDB().

3.19.1.2 long DBLinkTab_::size

Definition at line 95 of file database_util.h.

Referenced by DBlinkTabDestroy(), DBlinkTabEnd(), DBlinkTabInit(), DBlinkTabNew(), DBlinkTab-Remove(), DBlinkTabResize(), DBlinkTabSet(), and DBprintDB().

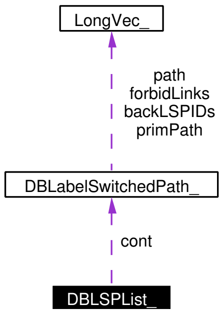
The documentation for this struct was generated from the following file:

- database_util.h

3.20 DBLSPList_ Struct Reference

`#include <database_st.h>`

Collaboration diagram for DBLSPList_:



Data Fields

- long size
- long top
- DBLabelSwitchedPath ** cont

3.20.1 Field Documentation

3.20.1.1 DBLabelSwitchedPath** DBLSPList_::cont

Definition at line 46 of file database_st.h.

Referenced by chooseReroutedLSPs(), DBlspListDestroy(), DBlspListEnd(), DBlspListInit(), DBlspList-Insert(), DBlspListNew(), DBlspListRemove(), and DBprintLink().

3.20.1.2 long DBLSPList_::size

Definition at line 44 of file database_st.h.

Referenced by DBlspListEnd(), DBlspListInit(), DBlspListInsert(), and DBlspListNew().

3.20.1.3 long DBLSPList_::top

Definition at line 45 of file database_st.h.

Referenced by chooseReroutedLSPs(), DBlspListEnd(), DBlspListInit(), DBlspListInsert(), DBlspList-New(), DBlspListRemove(), and DBprintLink().

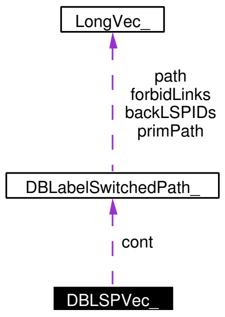
The documentation for this struct was generated from the following file:

- database_st.h

3.21 DBLSPVec_ Struct Reference

`#include <database_util.h>`

Collaboration diagram for DBLSPVec_:



Data Fields

- long size
- DBLabelSwitchedPath ** cont

3.21.1 Field Documentation

3.21.1.1 DBLabelSwitchedPath** DBLSPVec_::cont

Definition at line 76 of file database_util.h.

Referenced by DBlspVecDestroy(), DBlspVecEnd(), DBlspVecInit(), DBlspVecNew(), DBlspVec-Remove(), DBlspVecResize(), and DBlspVecSet().

3.21.1.2 long DBLSPVec_::size

Definition at line 75 of file database_util.h.

Referenced by DBlspVecDestroy(), DBlspVecEnd(), DBlspVecInit(), DBlspVecNew(), DBlspVec-Remove(), DBlspVecResize(), and DBlspVecSet().

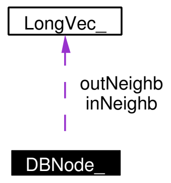
The documentation for this struct was generated from the following file:

- database_util.h

3.22 DBNode_ Struct Reference

`#include <database_util.h>`

Collaboration diagram for DBNode_:



Data Fields

- long id
- LongList inNeighb

  List of Nodes which have a link toward this Node.

- LongList outNeighb

  List of Nodes towards which this Node has a link.

3.22.1 Field Documentation

3.22.1.1 long DBNode_::id

Definition at line 16 of file database_util.h.

Referenced by computeBackup(), computeCost(), and DBaddNode().

3.22.1.2 LongList DBNode_::inNeighb

List of Nodes which have a link toward this Node.

Definition at line 18 of file database_util.h.

Referenced by DBaddLink(), DBgetNodeInNeighb(), DBnodeDestroy(), DBnodeEnd(), DBnodeInit(), DBnodeNew(), DBprintNode(), DBremoveLink(), and DBremoveNode().

3.22.1.3 LongList DBNode_::outNeighb

List of Nodes towards which this Node has a link.

Definition at line 20 of file database_util.h.

Referenced by DBaddLink(), DBgetNodeOutNeighb(), DBnodeDestroy(), DBnodeEnd(), DBnodeInit(), DBnodeNew(), DBprintNode(), DBremoveLink(), and DBremoveNode().

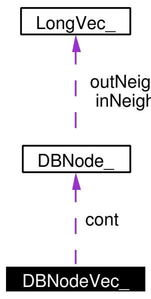
The documentation for this struct was generated from the following file:

- database_util.h

3.23 DBNodeVec_ Struct Reference

`#include <database_util.h>`

Collaboration diagram for DBNodeVec_:



Data Fields

- long size
- long top
- DBNode ** cont

3.23.1 Field Documentation

3.23.1.1 DBNode** DBNodeVec_::cont

Definition at line 57 of file database_util.h.

Referenced by computeBackup(), DBaddLink(), DBnodeVecDestroy(), DBnodeVecEnd(), DBnodeVecInit(), DBnodeVecNew(), DBnodeVecRemove(), DBnodeVecResize(), DBnodeVecSet(), DBprintDB(), and DBremoveLink().

3.23.1.2 long DBNodeVec_::size

Definition at line 55 of file database_util.h.

Referenced by DBnodeVecDestroy(), DBnodeVecEnd(), DBnodeVecInit(), DBnodeVecNew(), DBnodeVecRemove(), DBnodeVecResize(), DBnodeVecSet(), and DBprintDB().

3.23.1.3 long DBNodeVec_::top

Definition at line 56 of file database_util.h.

Referenced by computeBackup(), DBgetMaxNodeID(), DBnodeVecEnd(), DBnodeVecInit(), DBnodeVecNew(), DBnodeVecRemove(), DBnodeVecResize(), and DBnodeVecSet().

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

- database_util.h

3.24 DoubleVec_ Struct Reference

```
#include <common.h>
```

Data Fields

- long size
- long top
- double * cont

3.24.1 Field Documentation

3.24.1.1 double* DoubleVec_::cont

Definition at line 66 of file common.h.

Referenced by dblVecCopy(), dblVecDestroy(), dblVecEnd(), dblVecGet(), dblVecInit(), dblVecNew(), dblVecPopBack(), dblVecPushBack(), dblVecResize(), and dblVecSet().

3.24.1.2 long DoubleVec_::size

Definition at line 64 of file common.h.

Referenced by dblVecCopy(), dblVecEnd(), dblVecGet(), dblVecInit(), dblVecNew(), dblVecPushBack(), dblVecResize(), and dblVecSet().

3.24.1.3 long DoubleVec_::top

Definition at line 65 of file common.h.

Referenced by dblVecCopy(), dblVecEnd(), dblVecInit(), dblVecNew(), dblVecPopBack(), dblVecPush-Back(), and dblVecSet().

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

- common.h

3.25 ErrorElem_ Struct Reference

Data Fields

- GravityLevel gravity
- char message [ERRORMSG_SIZE]

3.25.1 Field Documentation

3.25.1.1 GravityLevel ErrorElem_::gravity

Definition at line 15 of file error.c.

Referenced by printErrorStack().

3.25.1.2 char ErrorElem_::message[ERRORMSG_SIZE]

Definition at line 16 of file error.c.

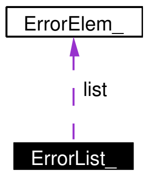
Referenced by printErrorStack().

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

- error.c

3.26 ErrorList_ Struct Reference

Collaboration diagram for ErrorList_:



Data Fields

- long size
- long top
- ErrorElem ∗ list

3.26.1 Field Documentation

3.26.1.1 ErrorElem∗ ErrorList_::list

Definition at line 23 of file error.c.

Referenced by addError(), errorDestroy(), errorInit(), and printErrorStack().

3.26.1.2 long ErrorList_::size

Definition at line 21 of file error.c.

Referenced by addError(), errorDestroy(), and errorInit().

3.26.1.3 long ErrorList_::top

Definition at line 22 of file error.c.

Referenced by addError(), errorDestroy(), errorInit(), and printErrorStack().

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

- error.c

3.27 libavl_allocator Struct Reference

```
#include <avl.h>
```

Data Fields

- void ∗(∗ libavl_malloc)(struct libavl_allocator ∗, size_t libavl_size)
- void(∗ libavl_free)(struct libavl_allocator ∗, void ∗libavl_block)

3.27.1 Field Documentation

3.27.1.1 void(∗ libavl_allocator::libavl_free)(struct libavl_allocator ∗, void ∗libavl_block)

Referenced by avl_destroy().

3.27.1.2 void∗(∗ libavl_allocator::libavl_malloc)(struct libavl_allocator ∗, size_t libavl_size)

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

- avl.h

3.28 LongVec_ Struct Reference

```
#include <common.h>
```

Data Fields

- long size
- long top
- long * cont

3.28.1 Field Documentation

3.28.1.1 long* LongVec_::cont

Definition at line 26 of file common.h.

Referenced by activateNodeInfo(), bellmanKalaba(), chooseReroutedLSPs(), colorClause(), computeBackup(), computeCost(), DBaddLink(), DBaddLSP(), DBprintNode(), DBremoveLink(), DBremoveNode(), fillTopo(), getRequestDst(), getRequestSrc(), initScore(), isValidRequestLink(), longListInsert(), longListMerge(), longListRemove(), longListSort(), longVecCopy(), longVecDestroy(), longVecEnd(), longVecGet(), longVecInit(), longVecNew(), longVecPopBack(), longVecPushBack(), longVecResize(), longVecSet(), makeScore(), noLoop(), printTopo(), updateLS(), updateNodeInfoOnElect(), and updateRequest().

3.28.1.2 long LongVec_::size

Definition at line 24 of file common.h.

Referenced by DBaddLink(), DBaddLSP(), longListInsert(), longVecCopy(), longVecEnd(), longVecGet(), longVecInit(), longVecNew(), longVecPushBack(), longVecResize(), and longVecSet().

3.28.1.3 long LongVec_::top

Definition at line 25 of file common.h.

Referenced by bellmanKalaba(), colorClause(), computeBackup(), computeCost(), DBaddLink(), DBaddLSP(), DBprintNode(), DBremoveLink(), DBremoveNode(), evalLS(), fillTopo(), getRequestDst(), getRequestSrc(), isValidRequestLink(), longListInsert(), longListMerge(), longListRemove(), longListSort(), longVecCopy(), longVecEnd(), longVecInit(), longVecNew(), longVecPopBack(), longVecPushBack(), longVecSet(), printTopo(), updateLS(), and updateRequest().

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

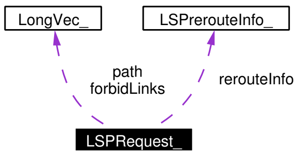
- common.h

3.29 LSPRequest_ Struct Reference

LSP Request Structure.

`#include <computation_st.h>`

Collaboration diagram for LSPRequest_:



Data Fields

- long id

  id of the LSP

- long primID

  id of the primary LSP if the LSP is a backup

- LSPrerouteInfo rerouteInfo
- int precedence

  preemption (or priority) level

- double bw [NB_OA]

  LSP bandwidth.

- LongList forbidLinks

  list of the link colors that can't be used

- LongList path

  path of the LSP

- DBLSPType type

  can be PRIM, GLOBAL_BACK or LOCAL_BACK

3.29.1 Detailed Description

LSP Request Structure.

Label Switched Path request representation, used by computePrimaryPath

Definition at line 28 of file computation_st.h.

3.29.2 Field Documentation

3.29.2.1 double LSPRequest_::bw[NB_OA]

LSP bandwidth.

Note that to be Diff-Serv Aware TE compliant, this field should be different from 0 only for one OA, because multiple OAs are not allowed on the same LSP

Definition at line 34 of file computation_st.h.

Referenced by computeBackup(), computeCost(), evalLS(), isValidLSPLink(), lspRequestInit(), lspRequestNew(), makeRerouteScore(), makeScore(), and updateNodeInfoOnElect().

3.29.2.2 LongList LSPRequest_::forbidLinks

list of the link colors that can't be used

Definition at line 35 of file computation_st.h.

Referenced by colorClause(), evalLS(), isValidLSPLink(), lspRequestCopy(), lspRequestDestroy(), lspRequestEnd(), lspRequestInit(), and lspRequestNew().

3.29.2.3 long LSPRequest_::id

id of the LSP

Definition at line 30 of file computation_st.h.

Referenced by evalLS(), isValidLSPLink(), isValidRequestLink(), and lspRequestCopy().

3.29.2.4 LongList LSPRequest_::path

path of the LSP

When using LSPRequest as an argument to computePrimaryPath, this list is filled with (src, -1, dst). When computePrimaryPath returns, this list contains the complete computed path

Definition at line 36 of file computation_st.h.

Referenced by computeBackup(), computeCost(), evalLS(), getRequestDst(), getRequestSrc(), isValidLSPLink(), isValidRequestLink(), lspRequestCopy(), lspRequestDestroy(), lspRequestEnd(), lspRequestInit(), lspRequestNew(), and updateRequest().

3.29.2.5 int LSPRequest_::precedence

preemption (or priority) level

Definition at line 33 of file computation_st.h.

Referenced by capacityClause(), computeBackup(), computeCost(), evalLS(), isValidLSPLink(), lspRequestCopy(), and makeRerouteScore().

3.29.2.6 long LSPRequest_::primID

id of the primary LSP if the LSP is a backup

Definition at line 31 of file computation_st.h.

Referenced by computeBackup(), computeCost(), evalLS(), isValidLSPLink(), and lspRequestCopy().

3.29.2.7 LSPrerouteInfo LSPRequest_::rerouteInfo

Definition at line 32 of file computation_st.h.

Referenced by evalLS(), isValidLSPLink(), isValidRequestLink(), lspRequestCopy(), lspRequestInit(), and lspRequestNew().

3.29.2.8 DBLSPType LSPRequest_::type

can be PRIM, GLOBAL_BACK or LOCAL_BACK

Definition at line 37 of file computation_st.h.

Referenced by computeBackup(), computeCost(), evalLS(), isValidLSPLink(), and lspRequestCopy().

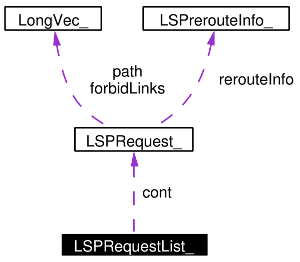
The documentation for this struct was generated from the following file:

- computation_st.h

3.30 LSPRequestList_ Struct Reference

`#include <computation_st.h>`

Collaboration diagram for LSPRequestList_:



Data Fields

- LSPRequest * cont
- long size

3.30.1 Field Documentation

3.30.1.1 LSPRequest* LSPRequestList_::cont

Definition at line 46 of file computation_st.h.

Referenced by lspRequestListEnd(), lspRequestListGet(), lspRequestListInit(), and lspRequestList-Resize().

3.30.1.2 long LSPRequestList_::size

Definition at line 47 of file computation_st.h.

Referenced by lspRequestListEnd(), lspRequestListGet(), lspRequestListInit(), lspRequestListResize(), and lspRequestListSize().

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

- computation_st.h

3.31 LSPrerouteInfo_ Struct Reference

Rerouting Information structure.

`#include <computation_st.h>`

Data Fields

- long id
  id of the preempted lsp
- long src
  the source of the link where preemption occurs
- long dst
  the destination of the link where preemption occurs

3.31.1 Detailed Description

Rerouting Information structure.

Used to support soft preemption. When a LSP is preempted, we have two choices. 1. Tear down this LSP immediately, this is hard preemption. 2. Notice the entity responsible for this LSP (e.g. the ingress in a decentralized mode) so that it can reestablish another LSP before the preempted one is being torn down. This is soft preemption. When soft preemption is used, when the computation of the new LSP (meant for replacing the soon preempted one) occurs, the computation algorithm must take into account the fact that the ressources of the preempted one can be used. But it is also interesting to take into account the link where the preemption occured, because it's certainly a link that must no more be used. In a decentralized approach, there's a good probability that the topology representation that the ingress has is not up-to-date when computing the rerouting. So, this is at least an interesting information to give to the computation algorithm.

Definition at line 17 of file computation_st.h.

3.31.2 Field Documentation

3.31.2.1 long LSPrerouteInfo_::dst

the destination of the link where preemption occurs

Definition at line 21 of file computation_st.h.

Referenced by isValidLSPLink(), and isValidRequestLink().

3.31.2.2 long LSPrerouteInfo_::id

id of the preempted lsp

That is the id of the lsp of which this lsp is the rerouting.

Definition at line 19 of file computation_st.h.

Referenced by evalLS(), isValidLSPLink(), isValidRequestLink(), lspRequestInit(), and lspRequestNew().

3.31.2.3 long LSPrerouteInfo_::src

the source of the link where preemption occurs

Definition at line 20 of file computation_st.h.

Referenced by isValidLSPLink(), and isValidRequestLink().

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

- computation_st.h

3.32 PredicateConfig_ Struct Reference

```
#include <setup.h>
```

Data Fields

- bool allowReroute
- bool capacityClause
- bool colorClause

3.32.1 Field Documentation

3.32.1.1 bool PredicateConfig_::allowReroute

Definition at line 89 of file setup.h.

Referenced by capacityClause().

3.32.1.2 bool PredicateConfig_::capacityClause

Definition at line 90 of file setup.h.

Referenced by isValidRequestLink().

3.32.1.3 bool PredicateConfig_::colorClause

Definition at line 91 of file setup.h.

Referenced by isValidRequestLink().

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

- setup.h

3.33 PrimaryComputationConfig_ Struct Reference

`#include <setup.h>`

Data Fields

- double loadBal [NB_OA]
- double load [NB_OA]
- double sqLoad [NB_OA]
- double relLoad [NB_OA]
- double sqRelLoad [NB_OA]
- double delay [NB_OA]
- double rerouting [NB_OA]

3.33.1 Field Documentation

3.33.1.1 double PrimaryComputationConfig_::delay[NB_OA]

Definition at line 83 of file setup.h.

Referenced by makeScore().

3.33.1.2 double PrimaryComputationConfig_::load[NB_OA]

Definition at line 79 of file setup.h.

Referenced by makeScore().

3.33.1.3 double PrimaryComputationConfig_::loadBal[NB_OA]

Definition at line 78 of file setup.h.

Referenced by activateNodeInfo(), initScore(), makeScore(), and updateNodeInfoOnElect().

3.33.1.4 double PrimaryComputationConfig_::relLoad[NB_OA]

Definition at line 81 of file setup.h.

Referenced by makeScore().

3.33.1.5 double PrimaryComputationConfig_::rerouting[NB_OA]

Definition at line 84 of file setup.h.

Referenced by makeScore().

3.33.1.6 double PrimaryComputationConfig_::sqLoad[NB_OA]

Definition at line 80 of file setup.h.

Referenced by makeScore().

3.33.1.7 double PrimaryComputationConfig_::sqRelLoad[NB_OA]

Definition at line 82 of file setup.h.

Referenced by makeScore().

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

- setup.h

3.34 ReroutingConfig_ Struct Reference

```
#include <setup.h>
```

Data Fields

- double scoreCoef [NB_OA][NB_PREEMPTION]

3.34.1 Field Documentation

3.34.1.1 double ReroutingConfig_::scoreCoef[NB_OA][NB_PREEMPTION]

Definition at line 96 of file setup.h.

Referenced by makeRerouteScore().

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

- setup.h

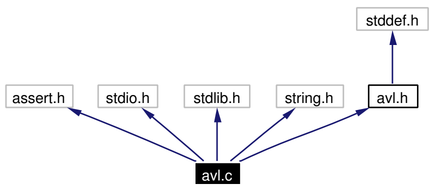
Chapter 4

DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering File Documentation

4.1 avl.c File Reference

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "avl.h"
```

Include dependency graph for avl.c:



Functions

- avl_table ∗ avl_create (avl_comparison_func ∗compare, void ∗param, struct libavl_allocator ∗allocator)
- void ∗ avl_find (const struct avl_table ∗tree, const void ∗item)
- void ∗∗ avl_probe (struct avl_table ∗tree, void ∗item)
- void ∗ avl_insert (struct avl_table ∗table, void ∗item)
- void ∗ avl_replace (struct avl_table ∗table, void ∗item)
- void ∗ avl_delete (struct avl_table ∗tree, const void ∗item)
- void avl_t_init (struct avl_traverser ∗trav, struct avl_table ∗tree)
- void ∗ avl_t_first (struct avl_traverser ∗trav, struct avl_table ∗tree)

- void * avl_t_last (struct avl_traverser *trav, struct avl_table *tree)
- void * avl_t_find (struct avl_traverser *trav, struct avl_table *tree, void *item)
- void * avl_t_insert (struct avl_traverser *trav, struct avl_table *tree, void *item)
- void * avl_t_copy (struct avl_traverser *trav, const struct avl_traverser *src)
- void * avl_t_next (struct avl_traverser *trav)
- void * avl_t_prev (struct avl_traverser *trav)
- void * avl_t_cur (struct avl_traverser *trav)
- void * avl_t_replace (struct avl_traverser *trav, void *new)
- avl_table * avl_copy (const struct avl_table *org, avl_copy_func *copy, avl_item_func *destroy, struct libavl_allocator *allocator)
- void avl_destroy (struct avl_table *tree, avl_item_func *destroy)
- void * avl_malloc (struct libavl_allocator *allocator, size_t size)
- void avl_free (struct libavl_allocator *allocator, void *block)

Variables

- libavl_allocator avl_allocator_default
- void(avl_assert_insert)(struct avl_table *table, void *item)
- void *(avl_assert_delete)(struct avl_table *table, void *item)

4.1.1 Function Documentation

4.1.1.1 struct avl_table* avl_copy (const struct avl_table * *org*, avl_copy_func * *copy*, avl_item_func * *destroy*, struct libavl_allocator * *allocator*)

Definition at line 727 of file avl.c.

References avl_table::avl_alloc, avl_node::avl_balance, avl_table::avl_compare, avl_copy(), avl_copy_-func, avl_table::avl_count, avl_create(), avl_node::avl_data, avl_item_func, avl_node::avl_link, AVL_MAX_-HEIGHT, and avl_table::avl_param.

Referenced by avl_copy().

```
{
  struct avl_node *stack[2 * (AVL_MAX_HEIGHT + 1)];
  int height = 0;

  struct avl_table *new;
  const struct avl_node *x;
  struct avl_node *y;

  assert (org != NULL);
  new = avl_create (org->avl_compare, org->avl_param,
                    allocator != NULL ? allocator : org->avl_alloc);
  if (new == NULL)
    return NULL;
  new->avl_count = org->avl_count;
  if (new->avl_count == 0)
    return new;

  x = (const struct avl_node *) &org->avl_root;
  y = (struct avl_node *) &new->avl_root;
  for (;;)
    {
      while (x->avl_link[0] != NULL)
        {
          assert (height < 2 * (AVL_MAX_HEIGHT + 1));

          y->avl_link[0] =
            new->avl_alloc->libavl_malloc (new->avl_alloc,
                                           sizeof *y->avl_link[0]);
          if (y->avl_link[0] == NULL)
            {
              if (y != (struct avl_node *) &new->avl_root)
                {
                  y->avl_data = NULL;
                  y->avl_link[1] = NULL;
                }

              copy_error_recovery (stack, height, new, destroy);
              return NULL;
            }

          stack[height++] = (struct avl_node *) x;
          stack[height++] = y;
          x = x->avl_link[0];
          y = y->avl_link[0];
        }
      y->avl_link[0] = NULL;

      for (;;)
        {
          y->avl_balance = x->avl_balance;
          if (copy == NULL)
            y->avl_data = x->avl_data;
          else
            {
              y->avl_data = copy (x->avl_data, org->avl_param);
              if (y->avl_data == NULL)
                {
                  y->avl_link[1] = NULL;
                  copy_error_recovery (stack, height, new, destroy);
                  return NULL;
                }
            }

          if (x->avl_link[1] != NULL)
            {
              y->avl_link[1] =
                new->avl_alloc->libavl_malloc (new->avl_alloc,
                                               sizeof *y->avl_link[1]);
              if (y->avl_link[1] == NULL)
                {
                  copy_error_recovery (stack, height, new, destroy);
                  return NULL;
                }

              x = x->avl_link[1];
              y = y->avl_link[1];
              break;
            }
          else
            y->avl_link[1] = NULL;

          if (height <= 2)
            return new;

          y = stack[--height];
          x = stack[--height];
        }
    }
}
```

4.1.1.2 struct avl_table∗ avl_create (avl_comparison_func ∗ *compare*, void ∗ *param*, struct libavl_allocator ∗ *allocator*)

Definition at line 37 of file avl.c.

References avl_table::avl_alloc, avl_allocator_default, avl_table::avl_compare, avl_comparison_func, avl_table::avl_count, avl_create(), avl_table::avl_generation, avl_table::avl_param, and avl_table::avl_root.

Referenced by avl_copy(), and avl_create().

```
{
  struct avl_table *tree;

  assert (compare != NULL);

  if (allocator == NULL)
    allocator = &avl_allocator_default;

  tree = allocator->libavl_malloc (allocator, sizeof *tree);
  if (tree == NULL)
    return NULL;

  tree->avl_root = NULL;
  tree->avl_compare = compare;
  tree->avl_param = param;
  tree->avl_alloc = allocator;
  tree->avl_count = 0;
  tree->avl_generation = 0;

  return tree;
}
```

4.1.1.3 void∗ avl_delete (struct avl_table ∗ *tree*, const void ∗ *item*)

Definition at line 228 of file avl.c.

References avl_node::avl_balance, avl_node::avl_data, avl_node::avl_link, and AVL_MAX_HEIGHT.

```
{
  /* Stack of nodes. */
  struct avl_node *pa[AVL_MAX_HEIGHT]; /* Nodes. */
  unsigned char da[AVL_MAX_HEIGHT];    /* |avl_link[]| indexes. */
  int k;                               /* Stack pointer. */

  struct avl_node *p;   /* Traverses tree to find node to delete. */
  int cmp;              /* Result of comparison between |item| and |p|. */

  assert (tree != NULL && item != NULL);

  k = 0;
  p = (struct avl_node *) &tree->avl_root;
  for (cmp = -1; cmp != 0;
       cmp = tree->avl_compare (item, p->avl_data, tree->avl_param))
    {
      int dir = cmp > 0;

      pa[k] = p;
      da[k++] = dir;

      p = p->avl_link[dir];
      if (p == NULL)
        return NULL;
    }
```

```
item = p->avl_data;

if (p->avl_link[1] == NULL)
  pa[k - 1]->avl_link[da[k - 1]] = p->avl_link[0];
else
  {
    struct avl_node *r = p->avl_link[1];
    if (r->avl_link[0] == NULL)
      {
        r->avl_link[0] = p->avl_link[0];
        r->avl_balance = p->avl_balance;
        pa[k - 1]->avl_link[da[k - 1]] = r;
        da[k] = 1;
        pa[k++] = r;
      }
    else
      {
        struct avl_node *s;
        int j = k++;

        for (;;)
          {
            da[k] = 0;
            pa[k++] = r;
            s = r->avl_link[0];
            if (s->avl_link[0] == NULL)
              break;

            r = s;
          }

        s->avl_link[0] = p->avl_link[0];
        r->avl_link[0] = s->avl_link[1];
        s->avl_link[1] = p->avl_link[1];
        s->avl_balance = p->avl_balance;

        pa[j - 1]->avl_link[da[j - 1]] = s;
        da[j] = 1;
        pa[j] = s;
      }
  }

tree->avl_alloc->libavl_free (tree->avl_alloc, p);

assert (k > 0);
while (--k > 0)
  {
    struct avl_node *y = pa[k];

    if (da[k] == 0)
      {
        y->avl_balance++;
        if (y->avl_balance == +1)
          break;
        else if (y->avl_balance == +2)
          {
            struct avl_node *x = y->avl_link[1];
            if (x->avl_balance == -1)
              {
                struct avl_node *w;
                assert (x->avl_balance == -1);
                w = x->avl_link[0];
                x->avl_link[0] = w->avl_link[1];
                w->avl_link[1] = x;
                y->avl_link[1] = w->avl_link[0];
                w->avl_link[0] = y;
                if (w->avl_balance == +1)
                  x->avl_balance = 0, y->avl_balance = -1;
                else if (w->avl_balance == 0)
                  x->avl_balance = y->avl_balance = 0;
                else /* |w->avl_balance == -1| */
                  x->avl_balance = +1, y->avl_balance = 0;
                w->avl_balance = 0;
                pa[k - 1]->avl_link[da[k - 1]] = w;
              }
            else
              {
                y->avl_link[1] = x->avl_link[0];
                x->avl_link[0] = y;
                pa[k - 1]->avl_link[da[k - 1]] = x;
                if (x->avl_balance == 0)
                  {
                    x->avl_balance = -1;
                    y->avl_balance = +1;
                    break;
                  }
                else
                  x->avl_balance = y->avl_balance = 0;
              }
          }
      }
    else
      {
        y->avl_balance--;
        if (y->avl_balance == -1)
          break;
        else if (y->avl_balance == -2)
          {
            struct avl_node *x = y->avl_link[0];
            if (x->avl_balance == +1)
              {
                struct avl_node *w;
                assert (x->avl_balance == +1);
                w = x->avl_link[1];
                x->avl_link[1] = w->avl_link[0];
                w->avl_link[0] = x;
                y->avl_link[0] = w->avl_link[1];
                w->avl_link[1] = y;
                if (w->avl_balance == -1)
                  x->avl_balance = 0, y->avl_balance = +1;
                else if (w->avl_balance == 0)
                  x->avl_balance = y->avl_balance = 0;
                else /* |w->avl_balance == +1| */
                  x->avl_balance = -1, y->avl_balance = 0;
                w->avl_balance = 0;
                pa[k - 1]->avl_link[da[k - 1]] = w;
              }
            else
              {
                y->avl_link[0] = x->avl_link[1];
                x->avl_link[1] = y;
                pa[k - 1]->avl_link[da[k - 1]] = x;
                if (x->avl_balance == 0)
                  {
                    x->avl_balance = +1;
                    y->avl_balance = -1;
                    break;
                  }
                else
                  x->avl_balance = y->avl_balance = 0;
              }
          }
      }
  }
```

```

  tree->avl_count--;
  tree->avl_generation++;
  return (void *) item;
}
```

4.1.1.4 void avl_destroy (struct avl_table ∗ *tree*, avl_item_func ∗ *destroy*)

Definition at line 822 of file avl.c.

References avl_table::avl_alloc, avl_node::avl_data, avl_node::avl_link, avl_table::avl_param, avl_-table::avl_root, and libavl_allocator::libavl_free.

```
{
  struct avl_node *p, *q;

  assert (tree != NULL);

  for (p = tree->avl_root; p != NULL; p = q)
    if (p->avl_link[0] == NULL)
      {
        q = p->avl_link[1];
        if (destroy != NULL && p->avl_data != NULL)
          destroy (p->avl_data, tree->avl_param);
        tree->avl_alloc->libavl_free (tree->avl_alloc, p);
      }
    else
      {
        q = p->avl_link[0];
        p->avl_link[0] = q->avl_link[1];
        q->avl_link[1] = p;
      }

  tree->avl_alloc->libavl_free (tree->avl_alloc, tree);
}
```

4.1.1.5 void∗ avl_find (const struct avl_table ∗ *tree*, const void ∗ *item*)

Definition at line 64 of file avl.c.

References avl_table::avl_compare, avl_node::avl_data, avl_node::avl_link, avl_table::avl_param, and avl_-table::avl_root.

```
{
  const struct avl_node *p;

  assert (tree != NULL && item != NULL);
  for (p = tree->avl_root; p != NULL; )
    {
      int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);

      if (cmp < 0)
        p = p->avl_link[0];
      else if (cmp > 0)
        p = p->avl_link[1];
      else /* |cmp == 0| */
        return p->avl_data;
    }

  return NULL;
}
```

4.1.1.6 void avl_free (struct libavl_allocator ∗ *allocator*, void ∗ *block*)

Definition at line 857 of file avl.c.

References free.

```
{
  assert (allocator != NULL && block != NULL);
  free (block);
}
```

4.1.1.7 void∗ avl_insert (struct avl_table ∗ *table*, void ∗ *item*)

Definition at line 201 of file avl.c.

References avl_probe().

```
{
  void **p = avl_probe (table, item);
  return p == NULL || *p == item ? NULL : *p;
}
```

4.1.1.8 void∗ avl_malloc (struct libavl_allocator ∗ *allocator*, size_t *size*)

Definition at line 849 of file avl.c.

References malloc.

```
{
  assert (allocator != NULL && size > 0);
  return malloc (size);
}
```

4.1.1.9 void∗∗ avl_probe (struct avl_table ∗ *tree*, void ∗ *item*)

Definition at line 89 of file avl.c.

References avl_node::avl_balance, avl_node::avl_data, avl_node::avl_link, and AVL_MAX_HEIGHT.

Referenced by avl_insert(), avl_replace(), and avl_t_insert().

```
{
  struct avl_node *y, *z; /* Top node to update balance factor, and parent. */
  struct avl_node *p, *q; /* Iterator, and parent. */
  struct avl_node *n;     /* Newly inserted node. */
  struct avl_node *w;     /* New root of rebalanced subtree. */
  int dir;                /* Direction to descend. */

  unsigned char da[AVL_MAX_HEIGHT]; /* Cached comparison results. */
  int k = 0;              /* Number of cached results. */

  assert (tree != NULL && item != NULL);

  z = (struct avl_node *) &tree->avl_root;
  y = tree->avl_root;
  dir = 0;
  for (q = z, p = y; p != NULL; q = p, p = p->avl_link[dir])
```

```
    {
      int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
      if (cmp == 0)
        return &p->avl_data;

      if (p->avl_balance != 0)
        z = q, y = p, k = 0;
      da[k++] = dir = cmp > 0;
    }

  n = q->avl_link[dir] =
    tree->avl_alloc->libavl_malloc (tree->avl_alloc, sizeof *n);
  if (n == NULL)
    return NULL;

  tree->avl_count++;
  n->avl_data = item;
  n->avl_link[0] = n->avl_link[1] = NULL;
  n->avl_balance = 0;
  if (y == NULL)
    return &n->avl_data;

  for (p = y, k = 0; p != n; p = p->avl_link[da[k]], k++)
    if (da[k] == 0)
      p->avl_balance--;
    else
      p->avl_balance++;

  if (y->avl_balance == -2)
    {
      struct avl_node *x = y->avl_link[0];
      if (x->avl_balance == -1)
        {
          w = x;
          y->avl_link[0] = x->avl_link[1];
          x->avl_link[1] = y;
          x->avl_balance = y->avl_balance = 0;
        }
      else
        {
          assert (x->avl_balance == +1);
          w = x->avl_link[1];
          x->avl_link[1] = w->avl_link[0];
          w->avl_link[0] = x;
          y->avl_link[0] = w->avl_link[1];
          w->avl_link[1] = y;
          if (w->avl_balance == -1)
            x->avl_balance = 0, y->avl_balance = +1;
          else if (w->avl_balance == 0)
            x->avl_balance = y->avl_balance = 0;
          else /* |w->avl_balance == +1| */
            x->avl_balance = -1, y->avl_balance = 0;
          w->avl_balance = 0;
        }
    }
  else if (y->avl_balance == +2)
    {
      struct avl_node *x = y->avl_link[1];
      if (x->avl_balance == +1)
        {
          w = x;
          y->avl_link[1] = x->avl_link[0];
          x->avl_link[0] = y;
          x->avl_balance = y->avl_balance = 0;
        }
      else
        {
          assert (x->avl_balance == -1);
          w = x->avl_link[0];
          x->avl_link[0] = w->avl_link[1];
          w->avl_link[1] = x;
          y->avl_link[1] = w->avl_link[0];
          w->avl_link[0] = y;
          if (w->avl_balance == +1)
            x->avl_balance = 0, y->avl_balance = -1;
          else if (w->avl_balance == 0)
            x->avl_balance = y->avl_balance = 0;
          else /* |w->avl_balance == -1| */
            x->avl_balance = +1, y->avl_balance = 0;
          w->avl_balance = 0;
        }
    }
  else
    return &n->avl_data;
  z->avl_link[y != z->avl_link[0]] = w;

  tree->avl_generation++;
  return &n->avl_data;
}
```

4.1.1.10 void∗ avl_replace (struct avl_table ∗ *table*, void ∗ *item*)

Definition at line 212 of file avl.c.

References avl_probe().

```
{
  void **p = avl_probe (table, item);
  if (p == NULL || *p == item)
    return NULL;
  else
    {
      void *r = *p;
      *p = item;
      return r;
    }
}
```

4.1.1.11 void∗ avl_t_copy (struct avl_traverser ∗ *trav*, const struct avl_traverser ∗ *src*)

Definition at line 557 of file avl.c.

References avl_node::avl_data, avl_traverser::avl_generation, avl_table::avl_generation, avl_traverser::avl_height, avl_traverser::avl_node, avl_traverser::avl_stack, and avl_traverser::avl_table.

```
{
  assert (trav != NULL && src != NULL);

  if (trav != src)
    {
      trav->avl_table = src->avl_table;
      trav->avl_node = src->avl_node;
      trav->avl_generation = src->avl_generation;
      if (trav->avl_generation == trav->avl_table->avl_generation)
        {
          trav->avl_height = src->avl_height;
          memcpy (trav->avl_stack, (const void *) src->avl_stack,
                  sizeof *trav->avl_stack * trav->avl_height);
        }
    }

  return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
}
```

4.1.1.12 void∗ avl_t_cur (struct avl_traverser ∗ *trav*)

Definition at line 685 of file avl.c.

References avl_node::avl_data, and avl_traverser::avl_node.

```
{
  assert (trav != NULL);

  return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
}
```

4.1.1.13 void∗ avl_t_find (struct avl_traverser ∗ *trav*, struct avl_table ∗ *tree*, void ∗ *item*)

Definition at line 493 of file avl.c.

References avl_table::avl_compare, avl_node::avl_data, avl_traverser::avl_generation, avl_table::avl_-generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_table::avl_param, avl_table::avl_root, avl_traverser::avl_stack, and avl_traverser::avl_table.

```
{
  struct avl_node *p, *q;

  assert (trav != NULL && tree != NULL && item != NULL);
  trav->avl_table = tree;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;
  for (p = tree->avl_root; p != NULL; p = q)
    {
      int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);

      if (cmp < 0)
        q = p->avl_link[0];
      else if (cmp > 0)
        q = p->avl_link[1];
      else /* |cmp == 0| */
        {
          trav->avl_node = p;
          return p->avl_data;
        }

      assert (trav->avl_height < AVL_MAX_HEIGHT);
      trav->avl_stack[trav->avl_height++] = p;
    }

  trav->avl_height = 0;
  trav->avl_node = NULL;
  return NULL;
}
```

4.1.1.14 void∗ avl_t_first (struct avl_traverser ∗ *trav*, struct avl_table ∗ *tree*)

Definition at line 437 of file avl.c.

References avl_node::avl_data, avl_traverser::avl_generation, avl_table::avl_generation, avl_traverser::avl_-height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_table::avl_root, avl_-traverser::avl_stack, and avl_traverser::avl_table.

Referenced by avl_t_next().

```
{
  struct avl_node *x;

  assert (tree != NULL && trav != NULL);

  trav->avl_table = tree;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;

  x = tree->avl_root;
  if (x != NULL)
    while (x->avl_link[0] != NULL)
      {
        assert (trav->avl_height < AVL_MAX_HEIGHT);
        trav->avl_stack[trav->avl_height++] = x;
        x = x->avl_link[0];
      }
  trav->avl_node = x;

  return x != NULL ? x->avl_data : NULL;
}
```

4.1.1.15 void avl_t_init (struct avl_traverser ∗ *trav*, struct avl_table ∗ *tree*)

Definition at line 425 of file avl.c.

References avl_traverser::avl_generation, avl_table::avl_generation, avl_traverser::avl_height, avl_-traverser::avl_node, and avl_traverser::avl_table.

Referenced by avl_t_insert().

```
{
  trav->avl_table = tree;
  trav->avl_node = NULL;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;
}
```

4.1.1.16 void∗ avl_t_insert (struct avl_traverser ∗ *trav*, struct avl_table ∗ *tree*, void ∗ *item*)

Definition at line 532 of file avl.c.

References avl_traverser::avl_generation, avl_table::avl_generation, avl_traverser::avl_node, avl_probe(), avl_t_init(), and avl_traverser::avl_table.

```
{
  void **p;

  assert (trav != NULL && tree != NULL && item != NULL);

  p = avl_probe (tree, item);
  if (p != NULL)
    {
      trav->avl_table = tree;
      trav->avl_node =
        ((struct avl_node *)
         ((char *) p - offsetof (struct avl_node, avl_data)));
      trav->avl_generation = tree->avl_generation - 1;
      return *p;
    }
  else
    {
      avl_t_init (trav, tree);
      return NULL;
    }
}
```

4.1.1.17 void* avl_t_last (struct avl_traverser * *trav*, struct avl_table * *tree*)

Definition at line 464 of file avl.c.

References avl_node::avl_data, avl_traverser::avl_generation, avl_table::avl_generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_table::avl_root, avl_traverser::avl_stack, and avl_traverser::avl_table.

Referenced by avl_t_prev().

```
{
  struct avl_node *x;

  assert (tree != NULL && trav != NULL);

  trav->avl_table = tree;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;

  x = tree->avl_root;
  if (x != NULL)
    while (x->avl_link[1] != NULL)
      {
        assert (trav->avl_height < AVL_MAX_HEIGHT);
        trav->avl_stack[trav->avl_height++] = x;
        x = x->avl_link[1];
      }
  trav->avl_node = x;

  return x != NULL ? x->avl_data : NULL;
}
```

4.1.1.18 void* avl_t_next (struct avl_traverser * *trav*)

Definition at line 581 of file avl.c.

References avl_node::avl_data, avl_traverser::avl_generation, avl_table::avl_generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_traverser::avl_stack, avl_t_first(), and avl_traverser::avl_table.

```
{
  struct avl_node *x;

  assert (trav != NULL);

  if (trav->avl_generation != trav->avl_table->avl_generation)
```

```
    trav_refresh (trav);

  x = trav->avl_node;
  if (x == NULL)
    {
      return avl_t_first (trav, trav->avl_table);
    }
  else if (x->avl_link[1] != NULL)
    {
      assert (trav->avl_height < AVL_MAX_HEIGHT);
      trav->avl_stack[trav->avl_height++] = x;
      x = x->avl_link[1];

      while (x->avl_link[0] != NULL)
        {
          assert (trav->avl_height < AVL_MAX_HEIGHT);
          trav->avl_stack[trav->avl_height++] = x;
          x = x->avl_link[0];
        }
    }
  else
    {
      struct avl_node *y;

      do
        {
          if (trav->avl_height == 0)
            {
              trav->avl_node = NULL;
              return NULL;
            }

          y = x;
          x = trav->avl_stack[--trav->avl_height];
        }
      while (y == x->avl_link[1]);
    }
  trav->avl_node = x;

  return x->avl_data;
}
```

4.1.1.19 void* avl_t_prev (struct avl_traverser * *trav*)

Definition at line 634 of file avl.c.

References avl_node::avl_data, avl_traverser::avl_generation, avl_table::avl_generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_traverser::avl_stack, avl_t_last(), and avl_traverser::avl_table.

```
{
  struct avl_node *x;

  assert (trav != NULL);

  if (trav->avl_generation != trav->avl_table->avl_generation)
    trav_refresh (trav);

  x = trav->avl_node;
  if (x == NULL)
    {
      return avl_t_last (trav, trav->avl_table);
    }
  else if (x->avl_link[0] != NULL)
```

```
    {
      assert (trav->avl_height < AVL_MAX_HEIGHT);
      trav->avl_stack[trav->avl_height++] = x;
      x = x->avl_link[0];

      while (x->avl_link[1] != NULL)
        {
          assert (trav->avl_height < AVL_MAX_HEIGHT);
          trav->avl_stack[trav->avl_height++] = x;
          x = x->avl_link[1];
        }
    }
  else
    {
      struct avl_node *y;

      do
        {
          if (trav->avl_height == 0)
            {
              trav->avl_node = NULL;
              return NULL;
            }

          y = x;
          x = trav->avl_stack[--trav->avl_height];
        }
      while (y == x->avl_link[0]);
    }
  trav->avl_node = x;

  return x->avl_data;
}
```

4.1.1.20 void∗ avl_t_replace (struct avl_traverser ∗ *trav*, void ∗ *new*)

Definition at line 696 of file avl.c.

References avl_node::avl_data, and avl_traverser::avl_node.

```
{
  void *old;

  assert (trav != NULL && trav->avl_node != NULL && new != NULL);
  old = trav->avl_node->avl_data;
  trav->avl_node->avl_data = new;
  return old;
}
```

4.1.2 Variable Documentation

4.1.2.1 struct libavl_allocator avl_allocator_default

Initial value:

```
  {
    avl_malloc,
    avl_free
  }
```

Definition at line 864 of file avl.c.

Referenced by avl_create().

4.1.2.2 void∗(avl_assert_delete)(struct avl_table ∗table, void ∗item)

Definition at line 884 of file avl.c.

```
{
  void *p = avl_delete (table, item);
  assert (p != NULL);
  return p;
}
```

4.1.2.3 void(avl_assert_insert)(struct avl_table ∗table, void ∗item)

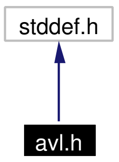
Definition at line 875 of file avl.c.

```
{
  void **p = avl_probe (table, item);
  assert (p != NULL && *p == item);
}
```

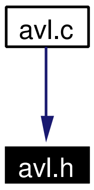
4.2 avl.h File Reference

`#include <stddef.h>`

Include dependency graph for avl.h:



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



Data Structures

- struct avl_node
- struct avl_table
- struct avl_traverser
- struct libavl_allocator

Defines

- #define AVL_H 1
- #define AVL_MAX_HEIGHT 32
- #define avl_count(table) ((size_t) (table) → avl_count)

Typedefs

- typedef int avl_comparison_func (const void ∗avl_a, const void ∗avl_b, void ∗avl_param)
- typedef void avl_item_func (void ∗avl_item, void ∗avl_param)
- typedef void ∗ avl_copy_func (void ∗avl_item, void ∗avl_param)

Functions

- void ∗ avl_malloc (struct libavl_allocator ∗, size_t)
- void avl_free (struct libavl_allocator ∗, void ∗)
- avl_table ∗ avl_create (avl_comparison_func ∗, void ∗, struct libavl_allocator ∗)

- avl_table ∗ avl_copy (const struct avl_table ∗, avl_copy_func ∗, avl_item_func ∗, struct libavl_allocator ∗)
- void avl_destroy (struct avl_table ∗, avl_item_func ∗)
- void ∗∗ avl_probe (struct avl_table ∗, void ∗)
- void ∗ avl_insert (struct avl_table ∗, void ∗)
- void ∗ avl_replace (struct avl_table ∗, void ∗)
- void ∗ avl_delete (struct avl_table ∗, const void ∗)
- void ∗ avl_find (const struct avl_table ∗, const void ∗)
- void avl_assert_insert (struct avl_table ∗, void ∗)
- void ∗ avl_assert_delete (struct avl_table ∗, void ∗)
- void avl_t_init (struct avl_traverser ∗, struct avl_table ∗)
- void ∗ avl_t_first (struct avl_traverser ∗, struct avl_table ∗)
- void ∗ avl_t_last (struct avl_traverser ∗, struct avl_table ∗)
- void ∗ avl_t_find (struct avl_traverser ∗, struct avl_table ∗, void ∗)
- void ∗ avl_t_insert (struct avl_traverser ∗, struct avl_table ∗, void ∗)
- void ∗ avl_t_copy (struct avl_traverser ∗, const struct avl_traverser ∗)
- void ∗ avl_t_next (struct avl_traverser ∗)
- void ∗ avl_t_prev (struct avl_traverser ∗)
- void ∗ avl_t_cur (struct avl_traverser ∗)
- void ∗ avl_t_replace (struct avl_traverser ∗, void ∗)

Variables

- libavl_allocator avl_allocator_default

4.2.1 Define Documentation

4.2.1.1 #define avl_count(table) ((size_t) (table) → avl_count)

Definition at line 101 of file avl.h.

4.2.1.2 #define AVL_H 1

Definition at line 27 of file avl.h.

4.2.1.3 #define AVL_MAX_HEIGHT 32

Definition at line 54 of file avl.h.

Referenced by avl_copy(), avl_delete(), avl_probe(), avl_t_find(), avl_t_first(), avl_t_last(), avl_t_next(), and avl_t_prev().

4.2.2 Typedef Documentation

4.2.2.1 typedef int avl_comparison_func(const void ∗avl_a, const void ∗avl_b, void ∗avl_param)

Definition at line 32 of file avl.h.

Referenced by avl_create().

4.2.2.2 typedef void∗ avl_copy_func(void ∗avl_item, void ∗avl_param)

Definition at line 35 of file avl.h.

Referenced by avl_copy().

4.2.2.3 typedef void avl_item_func(void ∗avl_item, void ∗avl_param)

Definition at line 34 of file avl.h.

Referenced by avl_copy().

4.2.3 Function Documentation

4.2.3.1 void∗ avl_assert_delete (struct avl_table ∗, void ∗)

4.2.3.2 void avl_assert_insert (struct avl_table ∗, void ∗)

4.2.3.3 struct avl_table∗ avl_copy (const struct avl_table ∗, avl_copy_func ∗, avl_item_func ∗, struct libavl_allocator ∗)

Definition at line 727 of file avl.c.

References avl_table::avl_alloc, avl_node::avl_balance, avl_table::avl_compare, avl_copy(), avl_copy_-func, avl_table::avl_count, avl_create(), avl_node::avl_data, avl_item_func, avl_node::avl_link, AVL_MAX_-HEIGHT, and avl_table::avl_param.

Referenced by avl_copy().

```
{
  struct avl_node *stack[2 * (AVL_MAX_HEIGHT + 1)];
  int height = 0;

  struct avl_table *new;
  const struct avl_node *x;
  struct avl_node *y;

  assert (org != NULL);
  new = avl_create (org->avl_compare, org->avl_param,
                    allocator != NULL ? allocator : org->avl_alloc);
  if (new == NULL)
    return NULL;
  new->avl_count = org->avl_count;
  if (new->avl_count == 0)
    return new;

  x = (const struct avl_node *) &org->avl_root;
  y = (struct avl_node *) &new->avl_root;
  for (;;)
    {
      while (x->avl_link[0] != NULL)
        {
          assert (height < 2 * (AVL_MAX_HEIGHT + 1));

          y->avl_link[0] =
            new->avl_alloc->libavl_malloc (new->avl_alloc,
                                           sizeof *y->avl_link[0]);
          if (y->avl_link[0] == NULL)
            {
              if (y != (struct avl_node *) &new->avl_root)
```

```
                {
                  y->avl_data = NULL;
                  y->avl_link[1] = NULL;
                }

              copy_error_recovery (stack, height, new, destroy);
              return NULL;
            }

          stack[height++] = (struct avl_node *) x;
          stack[height++] = y;
          x = x->avl_link[0];
          y = y->avl_link[0];
        }
      y->avl_link[0] = NULL;

      for (;;)
        {
          y->avl_balance = x->avl_balance;
          if (copy == NULL)
            y->avl_data = x->avl_data;
          else
            {
              y->avl_data = copy (x->avl_data, org->avl_param);
              if (y->avl_data == NULL)
                {
                  y->avl_link[1] = NULL;
                  copy_error_recovery (stack, height, new, destroy);
                  return NULL;
                }
            }

          if (x->avl_link[1] != NULL)
            {
              y->avl_link[1] =
                new->avl_alloc->libavl_malloc (new->avl_alloc,
                                               sizeof *y->avl_link[1]);
              if (y->avl_link[1] == NULL)
                {
                  copy_error_recovery (stack, height, new, destroy);
                  return NULL;
                }

              x = x->avl_link[1];
              y = y->avl_link[1];
              break;
            }
          else
            y->avl_link[1] = NULL;

          if (height <= 2)
            return new;

          y = stack[--height];
          x = stack[--height];
        }
    }
}
```

4.2.3.4 struct avl_table∗ avl_create (avl_comparison_func ∗, void ∗, struct libavl_allocator ∗)

Definition at line 37 of file avl.c.

References avl_table::avl_alloc, avl_allocator_default, avl_table::avl_compare, avl_comparison_func, avl_table::avl_count, avl_create(), avl_table::avl_generation, avl_table::avl_param, and avl_table::avl_root.

Referenced by avl_copy(), and avl_create().

```
{
  struct avl_table *tree;

  assert (compare != NULL);

  if (allocator == NULL)
    allocator = &avl_allocator_default;

  tree = allocator->libavl_malloc (allocator, sizeof *tree);
  if (tree == NULL)
    return NULL;

  tree->avl_root = NULL;
  tree->avl_compare = compare;
  tree->avl_param = param;
  tree->avl_alloc = allocator;
  tree->avl_count = 0;
  tree->avl_generation = 0;

  return tree;
}
```

4.2.3.5 void* avl_delete (struct avl_table *, const void *)

Definition at line 228 of file avl.c.

References avl_node::avl_balance, avl_node::avl_data, avl_node::avl_link, and AVL_MAX_HEIGHT.

```
{
  /* Stack of nodes. */
  struct avl_node *pa[AVL_MAX_HEIGHT]; /* Nodes. */
  unsigned char da[AVL_MAX_HEIGHT];    /* |avl_link[]| indexes. */
  int k;                               /* Stack pointer. */

  struct avl_node *p;   /* Traverses tree to find node to delete. */
  int cmp;              /* Result of comparison between |item| and |p|. */

  assert (tree != NULL && item != NULL);

  k = 0;
  p = (struct avl_node *) &tree->avl_root;
  for (cmp = -1; cmp != 0;
       cmp = tree->avl_compare (item, p->avl_data, tree->avl_param))
    {
      int dir = cmp > 0;

      pa[k] = p;
      da[k++] = dir;

      p = p->avl_link[dir];
      if (p == NULL)
        return NULL;
    }
  item = p->avl_data;

  if (p->avl_link[1] == NULL)
    pa[k - 1]->avl_link[da[k - 1]] = p->avl_link[0];
  else
    {
      struct avl_node *r = p->avl_link[1];
      if (r->avl_link[0] == NULL)
        {
            r->avl_link[0] = p->avl_link[0];
            r->avl_balance = p->avl_balance;
            pa[k - 1]->avl_link[da[k - 1]] = r;
            da[k] = 1;
            pa[k++] = r;
          }
        else
          {
            struct avl_node *s;
            int j = k++;

            for (;;)
              {
                da[k] = 0;
                pa[k++] = r;
                s = r->avl_link[0];
                if (s->avl_link[0] == NULL)
                  break;

                r = s;
              }

            s->avl_link[0] = p->avl_link[0];
            r->avl_link[0] = s->avl_link[1];
            s->avl_link[1] = p->avl_link[1];
            s->avl_balance = p->avl_balance;

            pa[j - 1]->avl_link[da[j - 1]] = s;
            da[j] = 1;
            pa[j] = s;
          }
      }

    tree->avl_alloc->libavl_free (tree->avl_alloc, p);

    assert (k > 0);
    while (--k > 0)
      {
        struct avl_node *y = pa[k];

        if (da[k] == 0)
          {
            y->avl_balance++;
            if (y->avl_balance == +1)
              break;
            else if (y->avl_balance == +2)
              {
                struct avl_node *x = y->avl_link[1];
                if (x->avl_balance == -1)
                  {
                    struct avl_node *w;
                    assert (x->avl_balance == -1);
                    w = x->avl_link[0];
                    x->avl_link[0] = w->avl_link[1];
                    w->avl_link[1] = x;
                    y->avl_link[1] = w->avl_link[0];
                    w->avl_link[0] = y;
                    if (w->avl_balance == +1)
                      x->avl_balance = 0, y->avl_balance = -1;
                    else if (w->avl_balance == 0)
                      x->avl_balance = y->avl_balance = 0;
                    else /* |w->avl_balance == -1| */
                      x->avl_balance = +1, y->avl_balance = 0;
                    w->avl_balance = 0;
                    pa[k - 1]->avl_link[da[k - 1]] = w;
                  }
                else
                  {
                    y->avl_link[1] = x->avl_link[0];
                    x->avl_link[0] = y;
                    pa[k - 1]->avl_link[da[k - 1]] = x;
                    if (x->avl_balance == 0)
                      {
                        x->avl_balance = -1;
                        y->avl_balance = +1;
                        break;
                      }
                    else
                      x->avl_balance = y->avl_balance = 0;
                  }
              }
          }
        else
          {
            y->avl_balance--;
            if (y->avl_balance == -1)
              break;
            else if (y->avl_balance == -2)
              {
                struct avl_node *x = y->avl_link[0];
                if (x->avl_balance == +1)
                  {
                    struct avl_node *w;
                    assert (x->avl_balance == +1);
                    w = x->avl_link[1];
                    x->avl_link[1] = w->avl_link[0];
                    w->avl_link[0] = x;
                    y->avl_link[0] = w->avl_link[1];
                    w->avl_link[1] = y;
                    if (w->avl_balance == -1)
                      x->avl_balance = 0, y->avl_balance = +1;
                    else if (w->avl_balance == 0)
                      x->avl_balance = y->avl_balance = 0;
                    else /* |w->avl_balance == +1| */
                      x->avl_balance = -1, y->avl_balance = 0;
                    w->avl_balance = 0;
                    pa[k - 1]->avl_link[da[k - 1]] = w;
                  }
                else
                  {
                    y->avl_link[0] = x->avl_link[1];
                    x->avl_link[1] = y;
                    pa[k - 1]->avl_link[da[k - 1]] = x;
                    if (x->avl_balance == 0)
                      {
                        x->avl_balance = +1;
                        y->avl_balance = -1;
                        break;
                      }
                    else
                      x->avl_balance = y->avl_balance = 0;
                  }
              }
          }
      }

    tree->avl_count--;
    tree->avl_generation++;
    return (void *) item;
  }
```

4.2.3.6 void avl_destroy (struct avl_table ∗, avl_item_func ∗)

Definition at line 822 of file avl.c.

References avl_table::avl_alloc, avl_node::avl_data, avl_node::avl_link, avl_table::avl_param, avl_-table::avl_root, and libavl_allocator::libavl_free.

```
{
  struct avl_node *p, *q;

  assert (tree != NULL);

  for (p = tree->avl_root; p != NULL; p = q)
    if (p->avl_link[0] == NULL)
      {
        q = p->avl_link[1];
        if (destroy != NULL && p->avl_data != NULL)
          destroy (p->avl_data, tree->avl_param);
        tree->avl_alloc->libavl_free (tree->avl_alloc, p);
      }
    else
      {
        q = p->avl_link[0];
        p->avl_link[0] = q->avl_link[1];
        q->avl_link[1] = p;
      }

  tree->avl_alloc->libavl_free (tree->avl_alloc, tree);
}
```

4.2.3.7 void∗ avl_find (const struct avl_table ∗, const void ∗)

Definition at line 64 of file avl.c.

References avl_table::avl_compare, avl_node::avl_data, avl_node::avl_link, avl_table::avl_param, and avl_-table::avl_root.

```
{
  const struct avl_node *p;

  assert (tree != NULL && item != NULL);
  for (p = tree->avl_root; p != NULL; )
    {
      int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);

      if (cmp < 0)
        p = p->avl_link[0];
      else if (cmp > 0)
        p = p->avl_link[1];
      else /* |cmp == 0| */
        return p->avl_data;
    }

  return NULL;
}
```

4.2.3.8 void avl_free (struct libavl_allocator ∗, void ∗)

Definition at line 857 of file avl.c.

References free.

```
{
  assert (allocator != NULL && block != NULL);
  free (block);
}
```

4.2.3.9 void∗ avl_insert (struct avl_table ∗, void ∗)

Definition at line 201 of file avl.c.

References avl_probe().

```
{
  void **p = avl_probe (table, item);
  return p == NULL || *p == item ? NULL : *p;
}
```

4.2.3.10 void∗ avl_malloc (struct libavl_allocator ∗, size_t)

Definition at line 849 of file avl.c.

References malloc.

```
{
  assert (allocator != NULL && size > 0);
  return malloc (size);
}
```

4.2.3.11 void∗∗ avl_probe (struct avl_table ∗, void ∗)

Definition at line 89 of file avl.c.

References avl_node::avl_balance, avl_node::avl_data, avl_node::avl_link, and AVL_MAX_HEIGHT.

Referenced by avl_insert(), avl_replace(), and avl_t_insert().

```
{
  struct avl_node *y, *z; /* Top node to update balance factor, and parent. */
  struct avl_node *p, *q; /* Iterator, and parent. */
  struct avl_node *n;     /* Newly inserted node. */
  struct avl_node *w;     /* New root of rebalanced subtree. */
  int dir;                /* Direction to descend. */

  unsigned char da[AVL_MAX_HEIGHT]; /* Cached comparison results. */
  int k = 0;              /* Number of cached results. */

  assert (tree != NULL && item != NULL);

  z = (struct avl_node *) &tree->avl_root;
  y = tree->avl_root;
  dir = 0;
  for (q = z, p = y; p != NULL; q = p, p = p->avl_link[dir])
    {
      int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
      if (cmp == 0)
        return &p->avl_data;

      if (p->avl_balance != 0)
        z = q, y = p, k = 0;
      da[k++] = dir = cmp > 0;
    }

  n = q->avl_link[dir] =
    tree->avl_alloc->libavl_malloc (tree->avl_alloc, sizeof *n);
  if (n == NULL)
    return NULL;

  tree->avl_count++;
  n->avl_data = item;
  n->avl_link[0] = n->avl_link[1] = NULL;
  n->avl_balance = 0;
  if (y == NULL)
    return &n->avl_data;

  for (p = y, k = 0; p != n; p = p->avl_link[da[k]], k++)
    if (da[k] == 0)
      p->avl_balance--;
    else
      p->avl_balance++;

  if (y->avl_balance == -2)
    {
      struct avl_node *x = y->avl_link[0];
      if (x->avl_balance == -1)
        {
          w = x;
          y->avl_link[0] = x->avl_link[1];
          x->avl_link[1] = y;
          x->avl_balance = y->avl_balance = 0;
        }
      else
        {
          assert (x->avl_balance == +1);
          w = x->avl_link[1];
          x->avl_link[1] = w->avl_link[0];
          w->avl_link[0] = x;
          y->avl_link[0] = w->avl_link[1];
          w->avl_link[1] = y;
          if (w->avl_balance == -1)
            x->avl_balance = 0, y->avl_balance = +1;
          else if (w->avl_balance == 0)
            x->avl_balance = y->avl_balance = 0;
          else /* |w->avl_balance == +1| */
            x->avl_balance = -1, y->avl_balance = 0;
          w->avl_balance = 0;
        }
    }
  else if (y->avl_balance == +2)
    {
      struct avl_node *x = y->avl_link[1];
      if (x->avl_balance == +1)
        {
          w = x;
          y->avl_link[1] = x->avl_link[0];
          x->avl_link[0] = y;
          x->avl_balance = y->avl_balance = 0;
        }
      else
        {
          assert (x->avl_balance == -1);
          w = x->avl_link[0];
          x->avl_link[0] = w->avl_link[1];
          w->avl_link[1] = x;
          y->avl_link[1] = w->avl_link[0];
          w->avl_link[0] = y;
          if (w->avl_balance == +1)
```

```
            x->avl_balance = 0, y->avl_balance = -1;
          else if (w->avl_balance == 0)
            x->avl_balance = y->avl_balance = 0;
          else /* |w->avl_balance == -1| */
            x->avl_balance = +1, y->avl_balance = 0;
          w->avl_balance = 0;
        }
    }
  else
    return &n->avl_data;
  z->avl_link[y != z->avl_link[0]] = w;

  tree->avl_generation++;
  return &n->avl_data;
}
```

4.2.3.12 void∗ avl_replace (struct avl_table ∗, void ∗)

Definition at line 212 of file avl.c.

References avl_probe().

```
{
  void **p = avl_probe (table, item);
  if (p == NULL || *p == item)
    return NULL;
  else
    {
      void *r = *p;
      *p = item;
      return r;
    }
}
```

4.2.3.13 void∗ avl_t_copy (struct avl_traverser ∗, const struct avl_traverser ∗)

Definition at line 557 of file avl.c.

References avl_node::avl_data, avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_height, avl_traverser::avl_node, avl_traverser::avl_stack, and avl_traverser::avl_table.

```
{
  assert (trav != NULL && src != NULL);

  if (trav != src)
    {
      trav->avl_table = src->avl_table;
      trav->avl_node = src->avl_node;
      trav->avl_generation = src->avl_generation;
      if (trav->avl_generation == trav->avl_table->avl_generation)
        {
          trav->avl_height = src->avl_height;
          memcpy (trav->avl_stack, (const void *) src->avl_stack,
                  sizeof *trav->avl_stack * trav->avl_height);
        }
    }

  return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
}
```

4.2.3.14 void∗ avl_t_cur (struct avl_traverser ∗)

Definition at line 685 of file avl.c.

References avl_node::avl_data, and avl_traverser::avl_node.

```
{
  assert (trav != NULL);

  return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
}
```

4.2.3.15 void∗ avl_t_find (struct avl_traverser ∗, struct avl_table ∗, void ∗)

Definition at line 493 of file avl.c.

References avl_table::avl_compare, avl_node::avl_data, avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_table::avl_param, avl_table::avl_root, avl_traverser::avl_stack, and avl_traverser::avl_table.

```
{
  struct avl_node *p, *q;

  assert (trav != NULL && tree != NULL && item != NULL);
  trav->avl_table = tree;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;
  for (p = tree->avl_root; p != NULL; p = q)
    {
      int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);

      if (cmp < 0)
        q = p->avl_link[0];
      else if (cmp > 0)
        q = p->avl_link[1];
      else /* |cmp == 0| */
        {
          trav->avl_node = p;
          return p->avl_data;
        }

      assert (trav->avl_height < AVL_MAX_HEIGHT);
      trav->avl_stack[trav->avl_height++] = p;
    }

  trav->avl_height = 0;
  trav->avl_node = NULL;
  return NULL;
}
```

4.2.3.16 void∗ avl_t_first (struct avl_traverser ∗, struct avl_table ∗)

Definition at line 437 of file avl.c.

References avl_node::avl_data, avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_table::avl_root, avl_traverser::avl_stack, and avl_traverser::avl_table.

Referenced by avl_t_next().

```
{
  struct avl_node *x;

  assert (tree != NULL && trav != NULL);

  trav->avl_table = tree;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;

  x = tree->avl_root;
  if (x != NULL)
    while (x->avl_link[0] != NULL)
      {
        assert (trav->avl_height < AVL_MAX_HEIGHT);
        trav->avl_stack[trav->avl_height++] = x;
        x = x->avl_link[0];
      }
  trav->avl_node = x;

  return x != NULL ? x->avl_data : NULL;
}
```

4.2.3.17 void avl_t_init (struct avl_traverser ∗, struct avl_table ∗)

Definition at line 425 of file avl.c.

References avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_height, avl_traverser::avl_node, and avl_traverser::avl_table.

Referenced by avl_t_insert().

```
{
  trav->avl_table = tree;
  trav->avl_node = NULL;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;
}
```

4.2.3.18 void∗ avl_t_insert (struct avl_traverser ∗, struct avl_table ∗, void ∗)

Definition at line 532 of file avl.c.

References avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_node, avl_probe(), avl_t_init(), and avl_traverser::avl_table.

```
{
  void **p;

  assert (trav != NULL && tree != NULL && item != NULL);

  p = avl_probe (tree, item);
  if (p != NULL)
    {
      trav->avl_table = tree;
      trav->avl_node =
        ((struct avl_node *)
         ((char *) p - offsetof (struct avl_node, avl_data)));
      trav->avl_generation = tree->avl_generation - 1;
      return *p;
    }
  else
```

```
    {
      avl_t_init (trav, tree);
      return NULL;
    }
}
```

4.2.3.19 void∗ avl_t_last (struct avl_traverser ∗, struct avl_table ∗)

Definition at line 464 of file avl.c.

References avl_node::avl_data, avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_table::avl_root, avl_traverser::avl_stack, and avl_traverser::avl_table.

Referenced by avl_t_prev().

```
{
  struct avl_node *x;

  assert (tree != NULL && trav != NULL);

  trav->avl_table = tree;
  trav->avl_height = 0;
  trav->avl_generation = tree->avl_generation;

  x = tree->avl_root;
  if (x != NULL)
    while (x->avl_link[1] != NULL)
      {
        assert (trav->avl_height < AVL_MAX_HEIGHT);
        trav->avl_stack[trav->avl_height++] = x;
        x = x->avl_link[1];
      }
  trav->avl_node = x;

  return x != NULL ? x->avl_data : NULL;
}
```

4.2.3.20 void∗ avl_t_next (struct avl_traverser ∗)

Definition at line 581 of file avl.c.

References avl_node::avl_data, avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_traverser::avl_stack, avl_t_first(), and avl_traverser::avl_table.

```
{
  struct avl_node *x;

  assert (trav != NULL);

  if (trav->avl_generation != trav->avl_table->avl_generation)
    trav_refresh (trav);

  x = trav->avl_node;
  if (x == NULL)
    {
      return avl_t_first (trav, trav->avl_table);
    }
  else if (x->avl_link[1] != NULL)
```

```
    {
      assert (trav->avl_height < AVL_MAX_HEIGHT);
      trav->avl_stack[trav->avl_height++] = x;
      x = x->avl_link[1];

      while (x->avl_link[0] != NULL)
        {
          assert (trav->avl_height < AVL_MAX_HEIGHT);
          trav->avl_stack[trav->avl_height++] = x;
          x = x->avl_link[0];
        }
    }
  else
    {
      struct avl_node *y;

      do
        {
          if (trav->avl_height == 0)
            {
              trav->avl_node = NULL;
              return NULL;
            }

          y = x;
          x = trav->avl_stack[--trav->avl_height];
        }
      while (y == x->avl_link[1]);
    }
  trav->avl_node = x;

  return x->avl_data;
}
```

4.2.3.21 void∗ avl_t_prev (struct avl_traverser ∗)

Definition at line 634 of file avl.c.

References avl_node::avl_data, avl_table::avl_generation, avl_traverser::avl_generation, avl_traverser::avl_-height, avl_node::avl_link, AVL_MAX_HEIGHT, avl_traverser::avl_node, avl_traverser::avl_stack, avl_t_-last(), and avl_traverser::avl_table.

```
{
  struct avl_node *x;

  assert (trav != NULL);

  if (trav->avl_generation != trav->avl_table->avl_generation)
    trav_refresh (trav);

  x = trav->avl_node;
  if (x == NULL)
    {
      return avl_t_last (trav, trav->avl_table);
    }
  else if (x->avl_link[0] != NULL)
    {
      assert (trav->avl_height < AVL_MAX_HEIGHT);
      trav->avl_stack[trav->avl_height++] = x;
      x = x->avl_link[0];

      while (x->avl_link[1] != NULL)
        {
          assert (trav->avl_height < AVL_MAX_HEIGHT);
          trav->avl_stack[trav->avl_height++] = x;
          x = x->avl_link[1];
        }
    }
  else
    {
      struct avl_node *y;

      do
        {
          if (trav->avl_height == 0)
            {
              trav->avl_node = NULL;
              return NULL;
            }

          y = x;
          x = trav->avl_stack[--trav->avl_height];
        }
      while (y == x->avl_link[0]);
    }
  trav->avl_node = x;

  return x->avl_data;
}
```

4.2.3.22 void∗ avl_t_replace (struct avl_traverser ∗, void ∗)

Definition at line 696 of file avl.c.

References avl_node::avl_data, and avl_traverser::avl_node.

```
{
  void *old;

  assert (trav != NULL && trav->avl_node != NULL && new != NULL);
  old = trav->avl_node->avl_data;
  trav->avl_node->avl_data = new;
  return old;
}
```

4.2.4 Variable Documentation

4.2.4.1 struct libavl_allocator avl_allocator_default

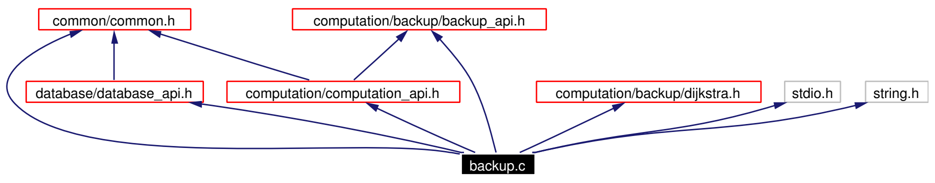
Definition at line 48 of file avl.h.

Referenced by avl_create().

4.3 backup.c File Reference

```
#include "computation/computation_api.h"
#include "computation/backup/backup_api.h"
#include "computation/backup/dijkstra.h"
#include "database/database_api.h"
#include "common/common.h"
#include <stdio.h>
#include <string.h>
```

Include dependency graph for backup.c:


common/common.h
computation/backup/backup_api.h
database/database_api.h
computation/computation_api.h
computation/backup/dijkstra.h
stdio.h
string.h
backup.c

Functions

- double computeCost (DataBase ∗dataBase, DBLinkState ∗∗lsList, long src, long dst, CPDijkNode ∗dn, DBLabelSwitchedPath ∗lsp, BackupType type)
- int computeBackup (DataBase ∗dataBase, LSPRequestList ∗reqList, long lspID, BackupType type)

  Backup LSP computation function.

4.3.1 Function Documentation

4.3.1.1 int computeBackup (DataBase ∗ *dataBase*, LSPRequestList ∗ *reqList*, long *lspID*, BackupType *type*)

Backup LSP computation function.

Parameters:

dataBase the general database containing topology

reqList the structure where computed backup lsp(s) will be stored

lspID the ID of the primary LSP for which local or global backup(s) are computed

type the type of the backup LSP(s) to be computed. If type is LOCAL_BACK, local backups will be computed, otherwise if type is GLOBAL_BACK, one global backup will be computed.

Definition at line 146 of file backup.c.

References addError(), LSPRequest_::bw, DBLabelSwitchedPath_::bw, calloc, computeCost(), DBNodeVec_::cont, LongVec_::cont, CPendPQ(), CPinitPQ(), CPinsertPQ(), CPpopTop(), CRITICAL, DBevalLSOnRemove(), DBevalLSOnSetup(), DBgetLinkDst(), DBgetLinkID(), DBgetLinkSrc(), DBgetLink-

State(), DBgetLSP(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), DBlinkStateCopy(), DBlinkState-Destroy(), DBlinkStateNew(), FALSE, free, CPDijkNode_::from, GLOBAL, GLOBAL_BACK, DBLabel-SwitchedPath_::id, DBNode_::id, INFO, DataBase_::linkSrcVec, LOCAL, LOCAL_BACK, longListEnd, longListInit, longListPushBack, lspRequestListGet(), lspRequestListResize(), CPDijkNode_::marked, NB_OA, NB_PREEMPTION, CPDijkNode_::node, DataBase_::nodeVec, NONE, DBLabelSwitchedPath_-::path, LSPRequest_::path, DBLabelSwitchedPath_::precedence, LSPRequest_::precedence, LSPRequest_-::primID, LongVec_::top, DBNodeVec_::top, TRUE, LSPRequest_::type, CPDijkNode_::val, and WARN-ING.

```
{
    long i,j,pNodeIndex,start,pNode;
    DBLinkState *ls, *oldLS;
    DBLinkState** lsList;
    CPDijkNode *dn=NULL;
    CPDijkNode** nodeList;
    bool reachPrimary;
    CPPrioQueue toBeTreated;
    LongList* neigh;
    LongList forbiddenLinks;
    DBLabelSwitchedPath* lsp;
    LSPRequest* req=NULL;
    double newVal;
    int pType=0;
    int src, dst;

    enum {NODE_FAILURE, LINK_FAILURE};

#if defined LINUX && defined TIMING && defined TIME3
    struct timezone tz;
    struct timeval  t1,t2;
#endif

    if (dataBase == NULL || reqList == NULL)
    {
        addError(CRITICAL,"Wrong argument in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (lspID < 0 || ((lsp = DBgetLSP(dataBase, lspID)) == NULL))
    {
        addError(CRITICAL,"Cannot find lsp ID in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lsList = calloc(dataBase->linkSrcVec.top, sizeof(DBLinkState*))) == NULL)
    {
        addError(CRITICAL,"Cannot allocate a required structure in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((nodeList = calloc(dataBase->nodeVec.top, sizeof(CPDijkNode*))) == NULL)
    {
        addError(CRITICAL,"Cannot allocate a required structure in %s at line %d",
                 __FILE__,__LINE__);
        free (lsList);
        return -1;
    }

    // duplicate all the link-states
    for (i=0; i<dataBase->linkSrcVec.top; i++)
    {
        src = DBgetLinkSrc(dataBase, i);
        dst = DBgetLinkDst(dataBase, i);

        if (src != -1 && dst != -1)
        {
            if ((oldLS = DBgetLinkState(dataBase, src, dst)) == NULL)
            {
                addError(WARNING,"Oups there should be a link-state here in %s at line %d",
                         __FILE__,__LINE__);
                continue;
            }

            if ((ls = DBlinkStateNew()) == NULL)
            {
                addError(CRITICAL,"Cannot duplicate all the link-states in %s at line %d",
                         __FILE__,__LINE__);
                continue;
            }

            if (DBlinkStateCopy(ls, oldLS) < 0)
            {
                addError(CRITICAL,"Something went wrong while copying in %s at line %d",
                 __FILE__,__LINE__);
                DBlinkStateDestroy(ls);
                continue;
            }

            lsList[i] = ls;
        }
        else
        {
            addError(INFO,"Warning there is no link numbered %ld : src = %ld, dst = %ld .... in %s at
                     __FILE__, __LINE__);
        }
    }

    // create the Dijk Nodes ... used for the computation
    for (i=0; i<dataBase->nodeVec.top; i++)
    {
        if (dataBase->nodeVec.cont[i] != NULL)
        {
            if ((dn = calloc(1,sizeof(CPDijkNode))) == NULL)
            {
                addError(CRITICAL,"Cannot create the Dijk nodes in %s at line %d",
                         __FILE__,__LINE__);
                continue;
            }

            dn->node = dataBase->nodeVec.cont[i];
            dn->val = -1;
            dn->marked = FALSE;

            nodeList[i] = dn;
        }
    }

    printf("Primary path :");
    for (i=0; i<lsp->path.top - 1; ++i)
    {
        printf("%ld - ", lsp->path.cont[i]);
    }
    printf("%ld\n", lsp->path.cont[i]);

    // now start the calculation ....

#if defined LINUX && defined TIMING && defined TIME3
    gettimeofday(&t1, &tz);
#endif
```

```

    // init a PrioQueue
    CPinitPQ(&toBeTreated);

    // init the forbiddenLinks;
    longListInit(&forbiddenLinks, -1);

    if (type == LOCAL)
    {
        // init the list of request to return;
        lspRequestListResize(reqList, lsp->path.top-1);
        for (i=0; i<lsp->path.top-1; ++i)
        {
            req = lspRequestListGet(reqList, i);

            req->primID = lsp->id;
            req->type = LOCAL_BACK;
            if (lsp->precedence + 1 < NB_PREEMPTION)
                req->precedence = lsp->precedence + 1;
            else
                req->precedence = lsp->precedence;
            memmove(&(req->bw),&(lsp->bw), NB_OA * sizeof(double));
        }

        // for (pNodeIndex=1; pNodeIndex<req->path.top; ++pNodeIndex) // start at 1 because we cannot
        for (pNodeIndex=lsp->path.top - 1; pNodeIndex>0; --pNodeIndex)
        {
            pNode = lsp->path.cont[pNodeIndex];
            start = lsp->path.cont[pNodeIndex - 1];

            pType = NODE_FAILURE;

            // mark the forbidden links
            forbiddenLinks.top = 0;

            if (pType == NODE_FAILURE)
            {
                neigh = DBgetNodeInNeighb(dataBase, pNode);
                if (neigh == NULL)
                {
                    addError(CRITICAL,"The protected node must have some neighbour in %s at line %d",
                             __FILE__,__LINE__);
                return -1;
                }

                for (i=0; i<neigh->top; ++i)
                {
                    longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[i], pNode));
                }
            }
            else if (pType == LINK_FAILURE)
            {
                longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));
            }

            // clear the PQ;
            while (CPpopTop(&toBeTreated) != NULL);

            // clear all the marks
            for (i=0; i<dataBase->nodeVec.top; i++)
            {
                nodeList[i]->marked = FALSE;
                nodeList[i]->val = -1;
                nodeList[i]->from = NULL;
            }

            // push the first node on the PQ
            CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

            reachPrimary = FALSE;
            while (reachPrimary == FALSE)
            {
                if ((dn = CPpopTop(&toBeTreated)) == NULL)
                {
                    // Oups ... impossible to reach the primary
                    // if we are in node protection mode, switch back to link protection
                    if (pType == NODE_FAILURE)
                    {
                        pType = LINK_FAILURE;

                        // mark the forbidden links
                        forbiddenLinks.top = 0;
                        longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));

                        // clear all marked nodes
                        for (i=0; i<dataBase->nodeVec.top; i++)
                        {
                            nodeList[i]->marked = FALSE;
                            nodeList[i]->val = -1;
                            nodeList[i]->from = NULL;
                        }

                        // push the first node on the PQ
                        CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

                        // re-enter the loop
                        continue;
                    }
                    else
                    {
                        break;
                    }
                }

                // as we don’t remove marked node immediatelly we may encounter one now so we skip it
                if (dn->marked == TRUE)
                    continue;

                // mark the node
                dn->marked = TRUE;

                // check the stop condition
                for (i=pNodeIndex; i<lsp->path.top; ++i)
                    if (lsp->path.cont[i] == dn->node->id)
                    {
                        reachPrimary = TRUE;
                        break;
                    }

                // we have finished ... leave the while loop
                if (reachPrimary == TRUE)
                    break;

                // find the neighbours
                neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);

                if (neigh != NULL)
                {
                    for (i=0; i<neigh->top; ++i)
                    {
                        int id;
                        double cost;

                        // check if the node is not already marked
                        if (nodeList[neigh->cont[i]]->marked == TRUE)
                            continue;

                        // check if the link is valid
                        id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
                        for (j=0; j<forbiddenLinks.top; ++j)
                            if (forbiddenLinks.cont[j] == id)
                                break;

                        if (j != forbiddenLinks.top)
                            continue;

                        // ok now update the node ...
                        cost = computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn, lsp, ty
                        if (cost >= 0) {
                            newVal = dn->val + cost;

                            if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeLis
                            {
                                nodeList[neigh->cont[i]]->val = newVal;
                                nodeList[neigh->cont[i]]->from = dn;
                                CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
                            }
                        }
                    }
                }
            }

            if (reachPrimary == TRUE)
            {
                req = lspRequestListGet(reqList, pNodeIndex-1);

                // clear the previous link state modification
                for (i=0; i<req->path.top - 1; i++)
                {
                    int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                    DBevalLSOnRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
                                     lsList[lnk], lsList[lnk], req);
                }

                // clear the old path ...
                req->path.top = 0;

                // ok we found a path ...
                printf("Cost = %f, Path = ", dn->val);

                while (dn != NULL && dn->from != NULL)
                {
                    longListPushBack(&(req->path), dn->node->id);
                    dn = dn->from;
                }
                longListPushBack(&(req->path), dn->node->id);

                // revert the path
                for (i=0; i<(req->path.top + 1)/2; i++)
                {
                    int tmp = req->path.cont[i];
                    req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
                    req->path.cont[req->path.top - 1 - i] = tmp;
                }

                for (i=0; i<req->path.top - 1; i++)
                {
                    int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                    DBevalLSOnSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
                                    lsList[lnk], lsList[lnk], req);
                }


                for (i=0; i<req->path.top - 1; i++)
                {
                    printf("%ld - ", req->path.cont[i]);
                }
                printf("%ld\n", req->path.cont[i]);
            }
            else
            {
                // oups we have to reject the request ...


            }

        }
    }
    else if (type == GLOBAL)
    {

        // init the list of request to return;
        lspRequestListResize(reqList, 1); // should not be required !
        req = lspRequestListGet(reqList, 0);

        req->primID = lsp->id;
        req->type = GLOBAL_BACK;
        if (lsp->precedence + 1 <NB_PREEMPTION)
            req->precedence = lsp->precedence + 1;
        else
            req->precedence = lsp->precedence;
        memmove(&(req->bw),&(lsp->bw), NB_OA * sizeof(double));

        start = lsp->path.cont[0];

        pType = NODE_FAILURE;

        // mark the forbidden links
        forbiddenLinks.top = 0;

        if (pType == NODE_FAILURE)
        {
            // don’t remove first and last node !!!
            for (i=1; i<lsp->path.top-1; i++)
            {
                neigh = DBgetNodeInNeighb(dataBase, lsp->path.cont[i]);
                if (neigh == NULL)
                {
                    addError(CRITICAL,"The protected node must have some neighbour in %s at line %d",
                             __FILE__,__LINE__);
                    return -1;
                }

                for (j=0; j<neigh->top; ++j)
                {
                    longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[j], lsp->path.
                }
            }

            // last link in the path must be removed !!!
            longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[lsp->path.top-2], l

        }
        else if (pType == LINK_FAILURE)
        {
            for (i=1; i<lsp->path.top; i++)
            {
                longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], lsp->path
            }
        }

        // clear the PQ;
        while (CPpopTop(&toBeTreated) != NULL);

        // clear all the marks
        for (i=0; i<dataBase->nodeVec.top; i++)
        {
            nodeList[i]->marked = FALSE;
            nodeList[i]->val = -1;
            nodeList[i]->from = NULL;
        }

        // push the first node on the PQ
        CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

        reachPrimary = FALSE;
        while (reachPrimary == FALSE)
        {
            if ((dn = CPpopTop(&toBeTreated)) == NULL)
            {
                // Oups ... impossible to reach the primary
                // if we are in node protection mode, switch back to link protection
                if (pType == NODE_FAILURE)
                {
                    printf("Oups ... switching protection ...\n");

                    pType = LINK_FAILURE;

                    // mark the forbidden links
                    forbiddenLinks.top = 0;
                    for (i=1; i<lsp->path.top; i++)
                    {
                        longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], l
                    }

                    // clear all marked nodes
                    for (i=0; i<dataBase->nodeVec.top; i++)
                    {
                        nodeList[i]->marked = FALSE;
                        nodeList[i]->val = -1;
                        nodeList[i]->from = NULL;
                    }

                    // push the first node on the PQ
                    CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

                    // re-enter the loop
                    continue;
                }
                else
                {
                    printf("Oups ... no path found ...\n");
                    break;
                }
            }

            // as we don’t remove marked node immediatelly we may encounter one now so we skip it
            if (dn->marked == TRUE)
                continue;

            // mark the node
            dn->marked = TRUE;

            // check the stop condition
            if (lsp->path.cont[lsp->path.top-1] == dn->node->id)
```

```
                reachPrimary = TRUE;

            // we have finished ... leave the while loop
            if (reachPrimary == TRUE)
                break;

            // find the neighbours
            neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);

            if (neigh != NULL)
            {
                for (i=0; i<neigh->top; ++i)
                {
                    int id;

                    // check if the node is not already marked
                    if (nodeList[neigh->cont[i]]->marked == TRUE)
                        continue;

                    // check if the link is valid
                    id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
                    for (j=0; j<forbiddenLinks.top; ++j)
                        if (forbiddenLinks.cont[j] == id)
                            break;

                    if (j != forbiddenLinks.top)
                        continue;

                    // ok now update the node ...
                    newVal = dn->val + computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn,

                    if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeList[neigh-
                    {
                        nodeList[neigh->cont[i]]->val = newVal;
                        nodeList[neigh->cont[i]]->from = dn;
                        CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
                    }
                }
            }
        }

        if (reachPrimary == TRUE)
        {
            req = lspRequestListGet(reqList, 0);

            // clear the previous link state modification
            for (i=0; i<req->path.top - 1; i++)
            {
                int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                DBevalLSOnRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
                                 lsList[lnk], lsList[lnk], req);
            }

            // clear the old path ...
            req->path.top = 0;

            // ok we found a path ...
            printf("Cost = %f, Path = ", dn->val);

            while (dn != NULL && dn->from != NULL)
            {
                longListPushBack(&(req->path), dn->node->id);
                dn = dn->from;
            }
            longListPushBack(&(req->path), dn->node->id);

            // revert the path
            for (i=0; i<(req->path.top + 1)/2; i++)
            {
                int tmp = req->path.cont[i];
                req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
                req->path.cont[req->path.top - 1 - i] = tmp;
            }

            for (i=0; i<req->path.top - 1; i++)
            {
                int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                DBevalLSOnSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
                                lsList[lnk], lsList[lnk], req);
            }


            for (i=0; i<req->path.top - 1; i++)
            {
                printf("%ld - ", req->path.cont[i]);
            }
            printf("%ld\n", req->path.cont[i]);
        }
        else
        {
            // oups we have to reject the request ...


        }
    }
    else if (type == NONE)
    {
        addError(INFO,"Oups no backup were requested in %s at line %d",
                        __FILE__,__LINE__);
    }
    else
    {
        // error
        addError(WARNING,"Unknown backup type in %s at line %d",
                 __FILE__,__LINE__);
    }

#if defined LINUX && defined TIMING && defined TIME3
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time for calculation of backups paths : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif

    // Clean Up

    // clear the PrioQueue
    CPendPQ(&toBeTreated);

    // clear the forbiddenLinks
    longListEnd(&forbiddenLinks);

    // clear the local copy ...
    for (i=0; i<dataBase->linkSrcVec.top; i++)
    {
        if (lsList[i] != NULL)
        {
            if (DBlinkStateDestroy(lsList[i]) < 0)
            {
                addError(CRITICAL,"Something went wrong while clearing a structure in %s at line %d",
                         __FILE__,__LINE__);
            }
        }
    }

    if (lsList != NULL)
        free(lsList);

    // free the dijkNodes
    for (i=0; i<dataBase->nodeVec.top; i++)
    {
        if (nodeList[i] != NULL)
        {
            free(nodeList[i]);
        }
    }

    if (nodeList)
        free(nodeList);

    return 0;
}
```

4.3.1.2 double computeCost (DataBase ∗ *dataBase*, DBLinkState ∗∗ *lsList*, long *src*, long *dst*, CPDijkNode ∗ *dn*, DBLabelSwitchedPath ∗ *lsp*, BackupType *type*)

Definition at line 15 of file backup.c.

References addError(), DBLabelSwitchedPath_::bw, LSPRequest_::bw, DBLinkState_::cap, LongVec_::cont, CRITICAL, DBevalLSOnSetup(), DBgetLinkID(), DBlinkStateEnd(), DBlinkStateInit(), CPDijkNode_::from, GLOBAL, GLOBAL_BACK, DBNode_::id, DBLabelSwitchedPath_::id, LOCAL, LOCAL_BACK, longListPushBack, lspRequestEnd(), lspRequestInit(), NB_OA, NB_PREEMPTION, CPDijkNode_::node, LSPRequest_::path, LSPRequest_::precedence, DBLabelSwitchedPath_::precedence, LSPRequest_::primID, DBLabelSwitchedPath_::primPath, DBLinkState_::rbw, LongVec_::top, LSPRequest_::type, and WARNING.

Referenced by computeBackup().

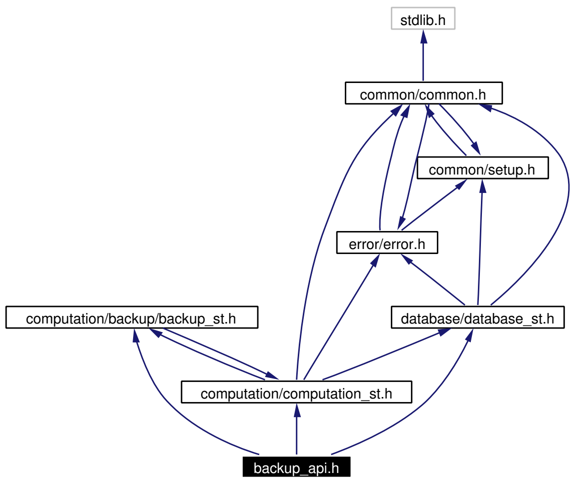
```
{
    CPDijkNode* ptr;
    LSPRequest newReq;
    DBLinkState newLS;
    long linkID,i,j;
    double alpha, beta;

    double inc;
    double bw_before[NB_OA];
    double bw_after[NB_OA];
    double bw_tot_bef=0, bw_tot_aft=0;
    double bw_tot=0;

    if (DBlinkStateInit(&newLS) < 0)
    {
        addError(CRITICAL,"Unable to init a link state in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    beta = 1.0/(1+10);
    alpha = 1 - beta;

    // bandwidth increment
    // -------------------

    lspRequestInit(&newReq);
    newReq.primID = lsp->id;

    if (lsp->precedence + 1 < NB_PREEMPTION)
```

```
        newReq.precedence = lsp->precedence + 1;
    else
        newReq.precedence = lsp->precedence;

    memmove(&(newReq.bw), &(lsp->bw), NB_OA * sizeof(double));

    if (type == LOCAL)
        newReq.type = LOCAL_BACK;
    else if (type == GLOBAL)
        newReq.type = GLOBAL_BACK;
    else
    {
        addError(WARNING,"Unknown backup type in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListPushBack(&(newReq.path), dst);

    ptr = dn;
    while (ptr != NULL)
    {
        longListPushBack(&(newReq.path), ptr->node->id);
        ptr = ptr->from;
    }

    // now reverse the path
    for (i=0; i<(newReq.path.top + 1)/2; i++)
    {
        int tmp = newReq.path.cont[i];
        newReq.path.cont[i] = newReq.path.cont[newReq.path.top - 1 - i];
        newReq.path.cont[newReq.path.top - 1 - i] = tmp;
    }

    // eval the impact of the addition
    if ((linkID = DBgetLinkID(dataBase, src, dst)) >= 0 &&
        DBevalLSOnSetup(dataBase, src, dst, &newLS, lsList[linkID], &newReq) >= 0)
    {
        for (i=0; i<NB_OA; ++i)
        {
            bw_tot += lsp->bw[i];

            bw_before[i] = 0;
            bw_after[i] = 0;
            for (j=0; j<NB_PREEMPTION; ++j)
            {
                bw_after[i] += newLS.rbw[i][j];
                bw_before[i] += lsList[linkID]->rbw[i][j];
            }
            bw_tot_bef += bw_before[i];
            bw_tot_aft += bw_after[i];
        }

        // printf("%ld-%ld : %f\n", src, dst, bw_after - bw_before);

    }
    else
    {
        addError(WARNING,"Error while computing new link state in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    lspRequestEnd(&newReq);

    DBlinkStateEnd(&newLS);

    // Capacity constrain ....
    for (i=0; i<NB_OA; i++)
        if (bw_after[i] > newLS.cap[i]) {
            return -1;
        }

    inc = bw_tot_aft - bw_tot_bef;
    if (inc < 0) {
        inc = 0;
    }

    // ------- Hop count ---------
    // ---------------------------

    // if we merge with the primary we have to account for the remaining of the path

    for (i=0; i<lsp->primPath.top; i++) {
        if (lsp->primPath.cont[i] == dst) {
            break;
        }
    }

    return (alpha * inc) + (beta * (lsp->primPath.top + 1 - i) * bw_tot);
}
```

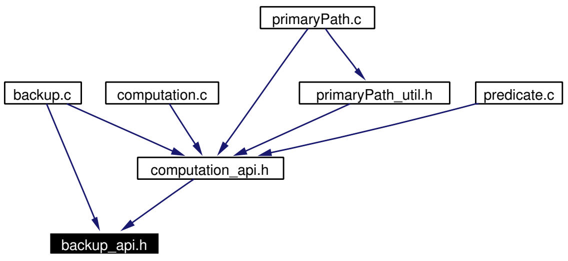
4.4 backup_api.h File Reference

```
#include "computation/backup/backup_st.h"
#include "database/database_st.h"
#include "computation/computation_st.h"
```

Include dependency graph for backup_api.h:



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



Functions

- int computeBackup (DataBase *dataBase, LSPRequestList *reqList, long lspID, BackupType type)

  Backup LSP computation function.

4.4.1 Function Documentation

4.4.1.1 int computeBackup (DataBase * *dataBase*, LSPRequestList * *reqList*, long *lspID*, BackupType *type*)

Backup LSP computation function.

Parameters:

dataBase the general database containing topology

reqList the structure where computed backup lsp(s) will be stored

lspID the ID of the primary LSP for which local or global backup(s) are computed

type the type of the backup LSP(s) to be computed. If type is LOCAL_BACK, local backups will be computed, otherwise if type is GLOBAL_BACK, one global backup will be computed.

Definition at line 146 of file backup.c.

References addError(), DBLabelSwitchedPath_::bw, LSPRequest_::bw, calloc, computeCost(), LongVec_::cont, DBNodeVec_::cont, CPendPQ(), CPinitPQ(), CPinsertPQ(), CPpopTop(), CRITICAL, DBevalLSOnRemove(), DBevalLSOnSetup(), DBgetLinkDst(), DBgetLinkID(), DBgetLinkSrc(), DBgetLinkState(), DBgetLSP(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateNew(), FALSE, free, CPDijkNode_::from, GLOBAL, GLOBAL_BACK, DBNode_::id, DBLabelSwitchedPath_::id, INFO, DataBase_::linkSrcVec, LOCAL, LOCAL_BACK, longListEnd, longListInit, longListPushBack, lspRequestListGet(), lspRequestListResize(), CPDijkNode_::marked, NB_OA, NB_PREEMPTION, CPDijkNode_::node, DataBase_::nodeVec, NONE, LSPRequest_::path, DBLabelSwitchedPath_::path, LSPRequest_::precedence, DBLabelSwitchedPath_::precedence, LSPRequest_::primID, DBNodeVec_::top, LongVec_::top, TRUE, LSPRequest_::type, CPDijkNode_::val, and WARNING.

```
{
    long i,j,pNodeIndex,start,pNode;
    DBLinkState *ls, *oldLS;
    DBLinkState** lsList;
    CPDijkNode *dn=NULL;
    CPDijkNode** nodeList;
    bool reachPrimary;
    CPPrioQueue toBeTreated;
    LongList* neigh;
    LongList forbiddenLinks;
    DBLabelSwitchedPath* lsp;
    LSPRequest* req=NULL;
    double newVal;
    int pType=0;
    int src, dst;

    enum {NODE_FAILURE, LINK_FAILURE};

#if defined LINUX && defined TIMING && defined TIME3
    struct timezone tz;
    struct timeval  t1,t2;
#endif

    if (dataBase == NULL || reqList == NULL)
    {
        addError(CRITICAL,"Wrong argument in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (lspID < 0 || ((lsp = DBgetLSP(dataBase, lspID)) == NULL))
    {
        addError(CRITICAL,"Cannot find lsp ID in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lsList = calloc(dataBase->linkSrcVec.top, sizeof(DBLinkState*))) == NULL)
    {
        addError(CRITICAL,"Cannot allocate a required structure in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((nodeList = calloc(dataBase->nodeVec.top, sizeof(CPDijkNode*))) == NULL)
    {
        addError(CRITICAL,"Cannot allocate a required structure in %s at line %d",
                 __FILE__,__LINE__);
        free (lsList);
        return -1;
    }

    // duplicate all the link-states
    for (i=0; i<dataBase->linkSrcVec.top; i++)
    {
        src = DBgetLinkSrc(dataBase, i);
        dst = DBgetLinkDst(dataBase, i);

        if (src != -1 && dst != -1)
        {
            if ((oldLS = DBgetLinkState(dataBase, src, dst)) == NULL)
            {
                addError(WARNING,"Oups there should be a link-state here in %s at line %d",
                         __FILE__,__LINE__);
                continue;
            }

            if ((ls = DBlinkStateNew()) == NULL)
            {
                addError(CRITICAL,"Cannot duplicate all the link-states in %s at line %d",
                         __FILE__,__LINE__);
                continue;
            }

            if (DBlinkStateCopy(ls, oldLS) < 0)
            {
                addError(CRITICAL,"Something went wrong while copying in %s at line %d",
                 __FILE__,__LINE__);
                DBlinkStateDestroy(ls);
                continue;
            }

            lsList[i] = ls;
        }
        else
        {
            addError(INFO,"Warning there is no link numbered %ld : src = %ld, dst = %ld .... in %s at
                     __FILE__, __LINE__);
        }
    }

    // create the Dijk Nodes ... used for the computation
    for (i=0; i<dataBase->nodeVec.top; i++)
    {
        if (dataBase->nodeVec.cont[i] != NULL)
        {
            if ((dn = calloc(1,sizeof(CPDijkNode))) == NULL)
            {
                addError(CRITICAL,"Cannot create the Dijk nodes in %s at line %d",
                         __FILE__,__LINE__);
                continue;
            }

            dn->node = dataBase->nodeVec.cont[i];
            dn->val = -1;
            dn->marked = FALSE;

            nodeList[i] = dn;
        }
    }

    printf("Primary path :");
    for (i=0; i<lsp->path.top - 1; ++i)
    {
        printf("%ld - ", lsp->path.cont[i]);
    }
    printf("%ld\n", lsp->path.cont[i]);

    // now start the calculation ....

#if defined LINUX && defined TIMING && defined TIME3
    gettimeofday(&t1, &tz);
#endif

    // init a PrioQueue
    CPinitPQ(&toBeTreated);

    // init the forbiddenLinks;
    longListInit(&forbiddenLinks, -1);

    if (type == LOCAL)
    {
        // init the list of request to return;
        lspRequestListResize(reqList, lsp->path.top-1);
        for (i=0; i<lsp->path.top-1; ++i)
        {
            req = lspRequestListGet(reqList, i);

            req->primID = lsp->id;
            req->type = LOCAL_BACK;
            if (lsp->precedence + 1 < NB_PREEMPTION)
                req->precedence = lsp->precedence + 1;
            else
                req->precedence = lsp->precedence;
            memmove(&(req->bw),&(lsp->bw), NB_OA * sizeof(double));
        }

        // for (pNodeIndex=1; pNodeIndex<req->path.top; ++pNodeIndex) // start at 1 because we cannot
        for (pNodeIndex=lsp->path.top - 1; pNodeIndex>0; --pNodeIndex)
        {
            pNode = lsp->path.cont[pNodeIndex];
            start = lsp->path.cont[pNodeIndex - 1];

            pType = NODE_FAILURE;

            // mark the forbidden links
            forbiddenLinks.top = 0;

            if (pType == NODE_FAILURE)
            {
                neigh = DBgetNodeInNeighb(dataBase, pNode);
                if (neigh == NULL)
                {
                    addError(CRITICAL,"The protected node must have some neighbour in %s at line %d",
                             __FILE__,__LINE__);
                return -1;
                }

                for (i=0; i<neigh->top; ++i)
                {
                    longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[i], pNode));
                }
            }
            else if (pType == LINK_FAILURE)
            {
```

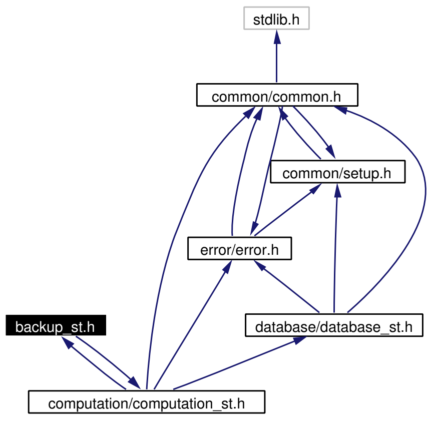
```
            longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));
        }

        // clear the PQ;
        while (CPpopTop(&toBeTreated) != NULL);

        // clear all the marks
        for (i=0; i<dataBase->nodeVec.top; i++)
        {
            nodeList[i]->marked = FALSE;
            nodeList[i]->val = -1;
            nodeList[i]->from = NULL;
        }

        // push the first node on the PQ
        CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

        reachPrimary = FALSE;
        while (reachPrimary == FALSE)
        {
            if ((dn = CPpopTop(&toBeTreated)) == NULL)
            {
                // Oups ... impossible to reach the primary
                // if we are in node protection mode, switch back to link protection
                if (pType == NODE_FAILURE)
                {
                    pType = LINK_FAILURE;

                    // mark the forbidden links
                    forbiddenLinks.top = 0;
                    longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));

                    // clear all marked nodes
                    for (i=0; i<dataBase->nodeVec.top; i++)
                    {
                        nodeList[i]->marked = FALSE;
                        nodeList[i]->val = -1;
                        nodeList[i]->from = NULL;
                    }

                    // push the first node on the PQ
                    CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

                    // re-enter the loop
                    continue;
                }
                else
                {
                    break;
                }
            }

            // as we don’t remove marked node immediatelly we may encounter one now so we skip it
            if (dn->marked == TRUE)
                continue;

            // mark the node
            dn->marked = TRUE;

            // check the stop condition
            for (i=pNodeIndex; i<lsp->path.top; ++i)
                if (lsp->path.cont[i] == dn->node->id)
                {
                    reachPrimary = TRUE;
                    break;
                }

            // we have finished ... leave the while loop
            if (reachPrimary == TRUE)
                break;

            // find the neighbours
            neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);

            if (neigh != NULL)
            {
                for (i=0; i<neigh->top; ++i)
                {
                    int id;
                    double cost;

                    // check if the node is not already marked
                    if (nodeList[neigh->cont[i]]->marked == TRUE)
                        continue;

                    // check if the link is valid
                    id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
                    for (j=0; j<forbiddenLinks.top; ++j)
                        if (forbiddenLinks.cont[j] == id)
                            break;

                    if (j != forbiddenLinks.top)
                        continue;

                    // ok now update the node ...
                    cost = computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn, lsp, ty
                    if (cost >= 0) {
                        newVal = dn->val + cost;

                        if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeLis
                        {
                            nodeList[neigh->cont[i]]->val = newVal;
                            nodeList[neigh->cont[i]]->from = dn;
                            CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
                        }
                    }
                }
            }
        }

        if (reachPrimary == TRUE)
        {
            req = lspRequestListGet(reqList, pNodeIndex-1);

            // clear the previous link state modification
            for (i=0; i<req->path.top - 1; i++)
            {
                int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                DBevalLSOnRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
                                 lsList[lnk], lsList[lnk], req);
            }

            // clear the old path ...
            req->path.top = 0;

            // ok we found a path ...
            printf("Cost = %f, Path = ", dn->val);

            while (dn != NULL && dn->from != NULL)
            {
                longListPushBack(&(req->path), dn->node->id);
                dn = dn->from;
            }
            longListPushBack(&(req->path), dn->node->id);
```

```

                // revert the path
                for (i=0; i<(req->path.top + 1)/2; i++)
                {
                    int tmp = req->path.cont[i];
                    req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
                    req->path.cont[req->path.top - 1 - i] = tmp;
                }

                for (i=0; i<req->path.top - 1; i++)
                {
                    int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                    DBevalLSOnSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
                                    lsList[lnk], lsList[lnk], req);
                }


                for (i=0; i<req->path.top - 1; i++)
                {
                    printf("%ld - ", req->path.cont[i]);
                }
                printf("%ld\n", req->path.cont[i]);
            }
            else
            {
                // oups we have to reject the request ...


            }

        }
    }
    else if (type == GLOBAL)
    {

        // init the list of request to return;
        lspRequestListResize(reqList, 1); // should not be required !
        req = lspRequestListGet(reqList, 0);

        req->primID = lsp->id;
        req->type = GLOBAL_BACK;
        if (lsp->precedence + 1 <NB_PREEMPTION)
            req->precedence = lsp->precedence + 1;
        else
            req->precedence = lsp->precedence;
        memmove(&(req->bw),&(lsp->bw), NB_OA * sizeof(double));

        start = lsp->path.cont[0];

        pType = NODE_FAILURE;

        // mark the forbidden links
        forbiddenLinks.top = 0;

        if (pType == NODE_FAILURE)
        {
            // don’t remove first and last node !!!
            for (i=1; i<lsp->path.top-1; i++)
            {
                neigh = DBgetNodeInNeighb(dataBase, lsp->path.cont[i]);
                if (neigh == NULL)
                {
                    addError(CRITICAL,"The protected node must have some neighbour in %s at line %d",
                             __FILE__,__LINE__);
                    return -1;
                }

                for (j=0; j<neigh->top; ++j)
                {
                    longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[j], lsp->path.
                }
            }

            // last link in the path must be removed !!!
            longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[lsp->path.top-2], l

        }
        else if (pType == LINK_FAILURE)
        {
            for (i=1; i<lsp->path.top; i++)
            {
                longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], lsp->path
            }
        }

        // clear the PQ;
        while (CPpopTop(&toBeTreated) != NULL);

        // clear all the marks
        for (i=0; i<dataBase->nodeVec.top; i++)
        {
            nodeList[i]->marked = FALSE;
            nodeList[i]->val = -1;
            nodeList[i]->from = NULL;
        }

        // push the first node on the PQ
        CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

        reachPrimary = FALSE;
        while (reachPrimary == FALSE)
        {
            if ((dn = CPpopTop(&toBeTreated)) == NULL)
            {
                // Oups ... impossible to reach the primary
                // if we are in node protection mode, switch back to link protection
                if (pType == NODE_FAILURE)
                {
                    printf("Oups ... switching protection ...\n");

                    pType = LINK_FAILURE;

                    // mark the forbidden links
                    forbiddenLinks.top = 0;
                    for (i=1; i<lsp->path.top; i++)
                    {
                        longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], l
                    }

                    // clear all marked nodes
                    for (i=0; i<dataBase->nodeVec.top; i++)
                    {
                        nodeList[i]->marked = FALSE;
                        nodeList[i]->val = -1;
                        nodeList[i]->from = NULL;
                    }

                    // push the first node on the PQ
                    CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));

                    // re-enter the loop
                    continue;
                }
                else
                {
                    printf("Oups ... no path found ...\n");
                    break;
                }
            }

            // as we don’t remove marked node immediatelly we may encounter one now so we skip it
            if (dn->marked == TRUE)
                continue;

            // mark the node
            dn->marked = TRUE;

            // check the stop condition
            if (lsp->path.cont[lsp->path.top-1] == dn->node->id)
                reachPrimary = TRUE;

            // we have finished ... leave the while loop
            if (reachPrimary == TRUE)
                break;

            // find the neighbours
            neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);

            if (neigh != NULL)
            {
                for (i=0; i<neigh->top; ++i)
                {
                    int id;

                    // check if the node is not already marked
                    if (nodeList[neigh->cont[i]]->marked == TRUE)
                        continue;

                    // check if the link is valid
                    id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
                    for (j=0; j<forbiddenLinks.top; ++j)
                        if (forbiddenLinks.cont[j] == id)
                            break;

                    if (j != forbiddenLinks.top)
                        continue;

                    // ok now update the node ...
                    newVal = dn->val + computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn,

                    if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeList[neigh-
                    {
                        nodeList[neigh->cont[i]]->val = newVal;
                        nodeList[neigh->cont[i]]->from = dn;
                        CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
                    }
                }
            }
        }

        if (reachPrimary == TRUE)
        {
            req = lspRequestListGet(reqList, 0);

            // clear the previous link state modification
            for (i=0; i<req->path.top - 1; i++)
            {
                int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                DBevalLSOnRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
                                 lsList[lnk], lsList[lnk], req);
            }

            // clear the old path ...
            req->path.top = 0;

            // ok we found a path ...
            printf("Cost = %f, Path = ", dn->val);

            while (dn != NULL && dn->from != NULL)
            {
                longListPushBack(&(req->path), dn->node->id);
                dn = dn->from;
            }
            longListPushBack(&(req->path), dn->node->id);

            // revert the path
            for (i=0; i<(req->path.top + 1)/2; i++)
            {
                int tmp = req->path.cont[i];
                req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
                req->path.cont[req->path.top - 1 - i] = tmp;
            }

            for (i=0; i<req->path.top - 1; i++)
            {
                int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
                DBevalLSOnSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
                                lsList[lnk], lsList[lnk], req);
            }


            for (i=0; i<req->path.top - 1; i++)
            {
                printf("%ld - ", req->path.cont[i]);
            }
            printf("%ld\n", req->path.cont[i]);
        }
        else
        {
            // oups we have to reject the request ...


        }
    }
    else if (type == NONE)
    {
        addError(INFO,"Oups no backup were requested in %s at line %d",
                         __FILE__,__LINE__);
    }
    else
    {
        // error
        addError(WARNING,"Unknown backup type in %s at line %d",
                 __FILE__,__LINE__);
    }

#if defined LINUX && defined TIMING && defined TIME3
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time for calculation of backups paths : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif

    // Clean Up

    // clear the PrioQueue
    CPendPQ(&toBeTreated);

    // clear the forbiddenLinks
    longListEnd(&forbiddenLinks);

    // clear the local copy ...
    for (i=0; i<dataBase->linkSrcVec.top; i++)
    {
        if (lsList[i] != NULL)
        {
            if (DBlinkStateDestroy(lsList[i]) < 0)
            {
                addError(CRITICAL,"Something went wrong while clearing a structure in %s at line %d",
                         __FILE__,__LINE__);
            }
        }
    }

    if (lsList != NULL)
        free(lsList);

    // free the dijkNodes
    for (i=0; i<dataBase->nodeVec.top; i++)
    {
        if (nodeList[i] != NULL)
        {
            free(nodeList[i]);
        }
    }

    if (nodeList)
        free(nodeList);

    return 0;
}
```

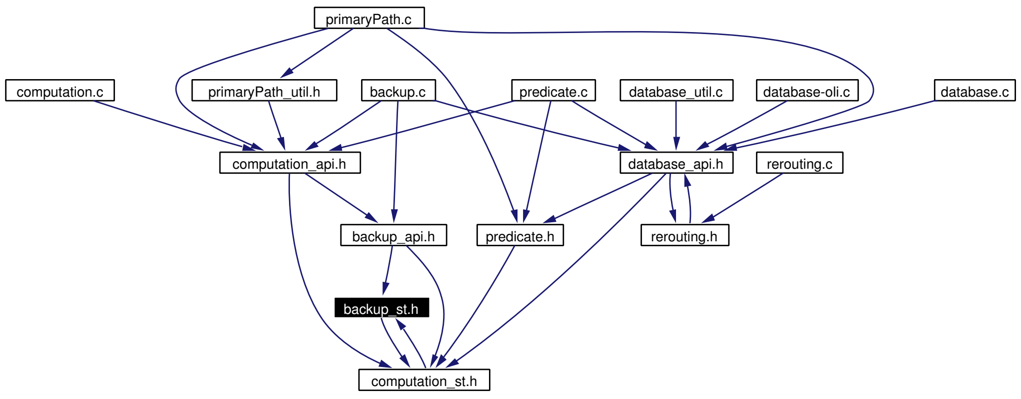
4.5 backup_st.h File Reference

```
#include "computation/computation_st.h"
```

Include dependency graph for backup_st.h:



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



Typedefs

- typedef unsigned char BackupType

Enumerations

- enum { NONE, LOCAL, GLOBAL }

4.5.1 Typedef Documentation

4.5.1.1 typedef unsigned char BackupType

Definition at line 7 of file backup_st.h.

4.5.2 Enumeration Type Documentation

4.5.2.1 anonymous enum

Enumeration values:

NONE

LOCAL

GLOBAL

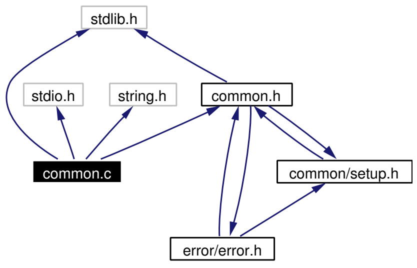
Definition at line 6 of file backup_st.h.

```
6 {NONE,LOCAL,GLOBAL};
```

4.6 common.c File Reference

```
#include "common.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Include dependency graph for common.c:



Functions

- LongVec ∗ longVecNew (long size)
- int longVecInit (LongVec ∗vec, long size)
- int longVecEnd (LongVec ∗vec)
- int longVecDestroy (LongVec ∗vec)
- int longVecCopy (LongVec ∗dst, LongVec ∗src)
- int longVecPushBack (LongVec ∗vec, long val)
- int longVecPopBack (LongVec ∗vec, long ∗val)
- int longVecResize (LongVec ∗vec, long newsize)
- int longVecGet (LongVec ∗vec, long index, long ∗val)
- int longVecSet (LongVec ∗vec, long index, long val)
- int longListInsert (LongList ∗list, long index, long val)
- int longListRemove (LongList ∗list, long index)
- int longCompare (const void ∗a, const void ∗b)
- int longListMerge (LongList ∗la, LongList ∗lb, LongList ∗dest)
- int longListSort (LongList ∗list)
- DoubleVec ∗ dblVecNew (long size)
- int dblVecInit (DoubleVec ∗vec, long size)
- int dblVecEnd (DoubleVec ∗vec)
- int dblVecDestroy (DoubleVec ∗vec)
- int dblVecCopy (DoubleVec ∗dst, DoubleVec ∗src)
- int dblVecPushBack (DoubleVec ∗vec, double val)
- int dblVecPopBack (DoubleVec ∗vec, double ∗val)
- int dblVecResize (DoubleVec ∗vec, long newsize)
- int dblVecGet (DoubleVec ∗vec, long index, double ∗val)
- int dblVecSet (DoubleVec ∗vec, long index, double val)
- void ∗ mymalloc (size_t sz)
- void ∗ myrealloc (void ∗ptr, size_t sz)
- void myfree (void ∗ptr)
- void ∗ mycalloc (size_t nmemb, size_t sz)

Variables

- long allocatedMemory = 0

4.6.1 Function Documentation

4.6.1.1 int dblVecCopy (DoubleVec ∗ *dst*, DoubleVec ∗ *src*)

Definition at line 529 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, realloc, DoubleVec_::size, and DoubleVec_::top.

Referenced by DBlinkStateCopy().

```
{
    double *ptr=NULL;

    if (dst == NULL || dst->cont == NULL ||
        src == NULL || src->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (dst->size < src->size)
    {
        if ((ptr=realloc(dst->cont,src->size*sizeof(double)))==NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            dst->cont=ptr;
            dst->size=src->size;
        }
    }

    memcpy(dst->cont,src->cont,src->size*sizeof(double));
    dst->top=src->top;

    return 0;
}
```

4.6.1.2 int dblVecDestroy (DoubleVec ∗ *vec*)

Definition at line 514 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, and free.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.6.1.3 int dblVecEnd (DoubleVec ∗ *vec*)

Definition at line 497 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, free, DoubleVec_::size, and DoubleVec_::top.

Referenced by DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), and DBlinkStateNew().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.6.1.4 int dblVecGet (DoubleVec ∗ *vec*, long *index*, double ∗ *val*)

Definition at line 640 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, and DoubleVec_::size.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[index];

    return 0;

}
```

4.6.1.5 int dblVecInit (DoubleVec ∗ *vec*, long *size*)

Definition at line 469 of file common.c.

References addError(), calloc, DoubleVec_::cont, CRITICAL, DBLVEC_INITSIZE, DoubleVec_::size, and DoubleVec_::top.

Referenced by DBlinkStateInit(), and DBlinkStateNew().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = DBLVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(double))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.6.1.6 DoubleVec∗ dblVecNew (long *size*)

Definition at line 439 of file common.c.

References addError(), calloc, DoubleVec_::cont, CRITICAL, DBLVEC_INITSIZE, free, DoubleVec_::size, and DoubleVec_::top.

```
{
    DoubleVec* vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(DoubleVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = DBLVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(double))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.6.1.7 int dblVecPopBack (DoubleVec ∗ *vec*, double ∗ *val*)

Definition at line 592 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, and DoubleVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[vec->top - 1];
    vec->top--;

    return 0;
}
```

4.6.1.8 int dblVecPushBack (DoubleVec ∗ *vec*, double *val*)

Definition at line 562 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, realloc, DoubleVec_::size, and DoubleVec_::top.

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if ((ptr = realloc(vec->cont, vec->size *
                           2 * sizeof(double))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        vec->size *= 2;
        vec->cont = ptr;
    }

    vec->cont[vec->top++] = val;

    return 0;
}
```

4.6.1.9 int dblVecResize (DoubleVec ∗ *vec*, long *newsize*)

Definition at line 615 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, realloc, and DoubleVec_::size.

Referenced by dblVecSet(), and updateLS().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = realloc(vec->cont, newsize*sizeof(double))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont = ptr;
    memset(ptr+ (vec->size * sizeof(double)), 0, (newsize - vec->size)*sizeof(double));
    vec->size = newsize;

    return 0;
}
```

4.6.1.10 int dblVecSet (DoubleVec ∗ *vec*, long *index*, double *val*)

Definition at line 669 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, dblVecResize(), max, DoubleVec_::size, and DoubleVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        if (dblVecResize(vec,max(vec->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize double vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    vec->cont[index] = val;
    vec->top = max(vec->top, index+1);

    return 0;
}
```

4.6.1.11 int longCompare (const void ∗ *a*, const void ∗ *b*)

Definition at line 347 of file common.c.

Referenced by longListMerge(), and longListSort().

```
{
    if ((*(long*)a)<(*(long*)b))
        return -1;
    else if ((*(long*)a)>(*(long*)b))
        return 1;
    else
        return 0;
}
```

4.6.1.12 int longListInsert (LongList ∗ *list*, long *index*, long *val*)

Definition at line 281 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longListResize, max, LongVec_::size, and LongVec_::top.

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((list->top >= list->size) || (index >= list->size))
    {
        if (longListResize(list,max(list->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize long vector in %s at line %d",
                    __FILE__,__LINE__);
            return -1;
        }
    }

    if (index < list->top)
    {
        memmove(list->cont+index+1,list->cont+index,(list->top-index) * sizeof(long));
    }
    list->cont[index]=val;
    list->top++;

    return 0;
}
```

4.6.1.13 int longListMerge (LongList ∗ *la*, LongList ∗ *lb*, LongList ∗ *dest*)

Definition at line 357 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longCompare(), longListCopy, longListEnd, longListInit, longListPushBack, and LongVec_::top.

Referenced by DBaddLSP().

```
{
    int i=0,j=0;
    LongList tmpList;

    if (la == NULL || la->cont == NULL ||
        lb == NULL || lb->cont == NULL ||
        dest == NULL || dest->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    qsort(la->cont,la->top,sizeof(long),&longCompare);
    qsort(lb->cont,lb->top,sizeof(long),&longCompare);

    if (longListInit(&tmpList,la->top+lb->top)<0)
    {
        addError(CRITICAL,"Unable to initialize temporary list of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    while ((i<la->top) || (j<lb->top))
    {
        if ((j>=lb->top) || (i<la->top && la->cont[i]<lb->cont[j]))
        {
            longListPushBack(&tmpList,la->cont[i]);
            i++;
        }
        else if ((i>=la->top) || (j<lb->top && la->cont[i]>lb->cont[j]))
        {
            longListPushBack(&tmpList,lb->cont[j]);
            j++;
        }
        else if (la->cont[i]==lb->cont[j])
        {
            longListPushBack(&tmpList,la->cont[i]);
            i++;
            j++;
        }
        else
        {
            addError(CRITICAL,"Internal error in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&tmpList);
            return -1;
        }
    }

    if (longListCopy(dest,&tmpList)<0)
    {
        addError(CRITICAL,"Unable to create resulting merged list of longs in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&tmpList);
        return -1;
    }

    longListEnd(&tmpList);

    return 0;
}
```

4.6.1.14 int longListRemove (LongList ∗ *list*, long *index*)

Definition at line 317 of file common.c.

References addError(), LongVec_::cont, CRITICAL, LongVec_::top, and WARNING.

Referenced by DBremoveLink().

```
    {
        if (list == NULL || list->cont == NULL)
        {
            addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        if (index < 0)
        {
            addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        if (index>=list->top)
        {
            addError(WARNING,"Removing inexistent list element in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            memmove(list->cont+index,list->cont+index+1,(list->top-index-1) * sizeof(long));
            list->top--;
            return 0;
        }
    }
```

4.6.1.15 int longListSort (LongList ∗ *list*)

Definition at line 421 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longCompare(), and LongVec_::top.

Referenced by DBaddLink().

```
                                  {

    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    qsort(list->cont,list->top,sizeof(long),&longCompare);

    return 0;
}
```

4.6.1.16 int longVecCopy (LongVec * *dst*, LongVec * *src*)

Definition at line 106 of file common.c.

References addError(), LongVec_::cont, CRITICAL, realloc, LongVec_::size, and LongVec_::top.

```
{
    long *ptr=NULL;

    if (dst == NULL || dst->cont == NULL ||
        src == NULL || src->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (dst->size < src->size)
    {
        if ((ptr=(long*) realloc(dst->cont,src->size*sizeof(long)))==NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            dst->cont=ptr;
            dst->size=src->size;
        }
    }

    memcpy(dst->cont,src->cont,src->size*sizeof(long));
    dst->top=src->top;

    return 0;
}
```

4.6.1.17 int longVecDestroy (LongVec * *vec*)

Definition at line 91 of file common.c.

References addError(), LongVec_::cont, CRITICAL, and free.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.6.1.18 int longVecEnd (LongVec ∗ *vec*)

Definition at line 74 of file common.c.

References addError(), LongVec_::cont, CRITICAL, free, LongVec_::size, and LongVec_::top.

Referenced by DBaddLSP(), DBdestroy(), DBnew(), and endTopo().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.6.1.19 int longVecGet (LongVec ∗ *vec*, long *index*, long ∗ *val*)

Definition at line 216 of file common.c.

References addError(), LongVec_::cont, CRITICAL, and LongVec_::size.

Referenced by DBgetLinkDst(), and DBgetLinkSrc().

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[index];

    return 0;

}
```

4.6.1.20 int longVecInit (LongVec ∗ *vec*, long *size*)

Definition at line 46 of file common.c.

References addError(), calloc, LongVec_::cont, CRITICAL, LONGVEC_INITSIZE, LongVec_::size, and LongVec_::top.

Referenced by DBaddLSP(), DBnew(), and initTopo().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LONGVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.6.1.21 LongVec∗ longVecNew (long *size*)

Definition at line 16 of file common.c.

References addError(), calloc, LongVec_::cont, CRITICAL, free, LONGVEC_INITSIZE, LongVec_::size, and LongVec_::top.

```
{
    LongVec* vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(LongVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LONGVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.6.1.22 int longVecPopBack (LongVec ∗ *vec*, long ∗ *val*)

Definition at line 169 of file common.c.

References addError(), LongVec_::cont, CRITICAL, and LongVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[vec->top - 1];
    vec->top--;

    return 0;
}
```

4.6.1.23 int longVecPushBack (LongVec ∗ *vec*, long *val*)

Definition at line 139 of file common.c.

References addError(), LongVec_::cont, CRITICAL, realloc, LongVec_::size, and LongVec_::top.

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if ((ptr = realloc(vec->cont, vec->size *
                           2 * sizeof(long))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        vec->size *= 2;
        vec->cont = ptr;
    }

    vec->cont[vec->top++] = val;

    return 0;
}
```

4.6.1.24 int longVecResize (LongVec ∗ *vec*, long *newsize*)

Definition at line 191 of file common.c.

References addError(), LongVec_::cont, CRITICAL, realloc, and LongVec_::size.

Referenced by longVecSet().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = realloc(vec->cont, newsize*sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont = ptr;
    memset(ptr+ (vec->size * sizeof(long)), 0, (newsize - vec->size)*sizeof(long));
    vec->size = newsize;

    return 0;
}
```

4.6.1.25 int longVecSet (LongVec ∗ *vec*, long *index*, long *val*)

Definition at line 245 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longVecResize(), max, LongVec_::size, and LongVec_::top.

Referenced by DBaddLink(), DBremoveLink(), and fillTopo().

```
246 {
247     if (vec == NULL || vec->cont == NULL)
248     {
249         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
250                  __FILE__,__LINE__);
251         return -1;
252     }
253 
254     if (index < 0)
255     {
256         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
257                  __FILE__,__LINE__);
258         return -1;
259     }
260 
261     if (index >= vec->size)
262     {
263         if (longVecResize(vec,max(vec->size * 2,index+1))<0)
264         {
265             addError(CRITICAL,"Unable to resize long vector in %s at line %d",
266                      __FILE__,__LINE__);
267             return -1;
268         }
269     }
270 
271     vec->cont[index] = val;
272     vec->top = max(vec->top, index+1);
273 
274     return 0;
275 }
```

4.6.1.26 void* mycalloc (size_t *nmemb*, size_t *sz*)

Definition at line 751 of file common.c.

References mymalloc().

```
752 {
753     void *ptr;
754 
755     if ((ptr=mymalloc(nmemb*sz))==NULL)
756         return NULL;
757 
758     memset(ptr,0,nmemb*sz);
759 
760     return ptr;
761 }
```

4.6.1.27 void myfree (void * *ptr*)

Definition at line 741 of file common.c.

References allocatedMemory, and free.

```
742 {
    if (*((long*)(ptr-sizeof(long))) == -1)
        fprintf(stderr, "Warning already freed\n");

    allocatedMemory-=*((long*)(ptr-sizeof(long)));
    *((long*)(ptr-sizeof(long)))=-1;
    free(ptr-sizeof(long));
}
```

4.6.1.28 void∗ mymalloc (size_t *sz*)

Definition at line 715 of file common.c.

References allocatedMemory, and malloc.

Referenced by mycalloc().

```
{
    void *ptr;

    if ((ptr=malloc(sz+sizeof(long)))==NULL)
        return NULL;

    allocatedMemory+=sz;
    *((long*)ptr)=sz;

    return (ptr+sizeof(long));
}
```

4.6.1.29 void∗ myrealloc (void ∗ *ptr*, size_t *sz*)

Definition at line 728 of file common.c.

References allocatedMemory, and realloc.

```
{
    void *retptr;

    if ((retptr=realloc(ptr-sizeof(long),sz+sizeof(long)))==NULL)
        return NULL;

    allocatedMemory+=sz-(*((long*)(retptr)));
    *((long*)retptr)=sz;

    return (retptr+sizeof(long));
}
```

4.6.2 Variable Documentation

4.6.2.1 long allocatedMemory = 0

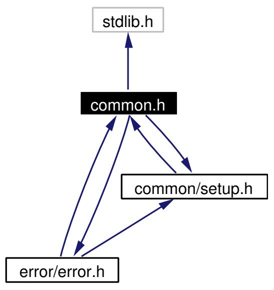
Definition at line 713 of file common.c.

Referenced by myfree(), mymalloc(), and myrealloc().

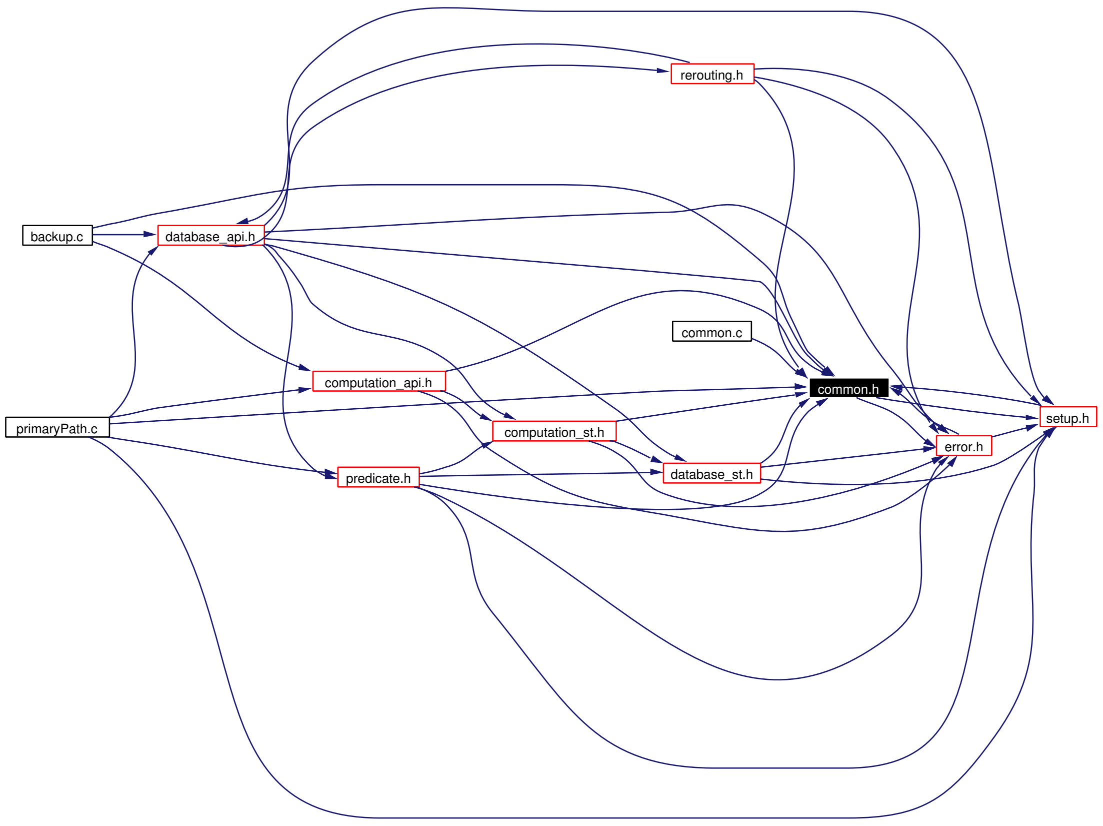
4.7 common.h File Reference

```
#include "error/error.h"
#include "common/setup.h"
#include <stdlib.h>
```

Include dependency graph for common.h:



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



Data Structures

- struct DoubleVec_
- struct LongVec_

Defines

- #define max(a, b) (((a)>(b))?(a):(b))
- #define min(a, b) (((a)<(b))?(a):(b))
- #define longListNew(a) longVecNew(a)
- #define longListInit(a, b) longVecInit((LongVec*) a,b)
- #define longListEnd(a) longVecEnd((LongVec*) a)
- #define longListDestroy(a) longVecDestroy((LongVec*) a)
- #define longListCopy(a, b) longVecCopy((LongVec*) a,(LongVec*) b)
- #define longListResize(a, b) longVecResize((LongVec*) a,b)
- #define longListPushBack(a, b) longVecPushBack((LongVec*) a,b)
- #define longListPopBack(a, b) longVecPopBack((LongVec*) a,b)
- #define malloc(a) mymalloc(a)
- #define realloc(a, b) myrealloc(a,b)
- #define free(a) myfree(a)
- #define calloc(a, b) mycalloc(a,b)
- #define ANDERROR(a, b) (a=((b)<0?-1:(a)))

Typedefs

- typedef unsigned char bool
- typedef LongVec_ LongVec
- typedef LongVec LongList
- typedef DoubleVec_ DoubleVec

Enumerations

- enum { FALSE = 0, TRUE = 1 }

Functions

- LongVec * longVecNew (long)
- int longVecInit (LongVec *, long)
- int longVecEnd (LongVec *)
- int longVecDestroy (LongVec *)
- int longVecCopy (LongVec *, LongVec *)
- int longVecPushBack (LongVec *, long)
- int longVecPopBack (LongVec *, long *)
- int longVecResize (LongVec *, long)
- int longVecGet (LongVec *, long, long *)
- int longVecSet (LongVec *, long, long)
- int longListInsert (LongList *, long, long)
- int longListRemove (LongList *, long)
- int longListMerge (LongList *, LongList *, LongList *)

- int longListSort (LongList ∗)
- DoubleVec ∗ dblVecNew (long)
- int dblVecInit (DoubleVec ∗, long)
- int dblVecEnd (DoubleVec ∗)
- int dblVecDestroy (DoubleVec ∗)
- int dblVecCopy (DoubleVec ∗, DoubleVec ∗)
- int dblVecPushBack (DoubleVec ∗, double)
- int dblVecPopBack (DoubleVec ∗, double ∗)
- int dblVecResize (DoubleVec ∗, long)
- int dblVecGet (DoubleVec ∗, long, double ∗)
- int dblVecSet (DoubleVec ∗, long, double)
- void ∗ mymalloc (size_t)
- void ∗ myrealloc (void ∗, size_t)
- void myfree (void ∗)
- void ∗ mycalloc (size_t, size_t)

Variables

- long allocatedMemory

4.7.1 Define Documentation

4.7.1.1 #define ANDERROR(a, b) (a=((b)<0?-1:(a)))

Definition at line 98 of file common.h.

Referenced by DBaddLink(), DBaddLSP(), DBlinkStateCopy(), DBlspCopy(), DBremoveLink(), DBremoveLSP(), DBremoveNode(), and lspRequestCopy().

4.7.1.2 #define calloc(a, b) mycalloc(a,b)

Definition at line 91 of file common.h.

Referenced by bellmanKalaba(), bkConnectVecInit(), bkNodeVecInit(), bkNodeVecNew(), computeBackup(), CPnewPQ(), CPnewTN(), DBlinkNew(), DBlinkStateNew(), DBlinkTabInit(), DBlinkTabNew(), DBlinkTabResize(), DBlspListInit(), DBlspListNew(), DBlspNew(), DBlspVecInit(), DBlspVecNew(), dblVecInit(), dblVecNew(), DBnew(), DBnodeNew(), DBnodeVecInit(), DBnodeVecNew(), errorInit(), fillTopo(), longVecInit(), longVecNew(), lspRequestListInit(), and lspRequestNew().

4.7.1.3 #define free(a) myfree(a)

Definition at line 90 of file common.h.

Referenced by avl_free(), bellmanKalaba(), bkConnectVecDestroy(), bkConnectVecEnd(), bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecInit(), bkNodeVecNew(), computeBackup(), CPdestroyPQ(), CPdestroyTN(), DBdestroy(), DBlinkDestroy(), DBlinkNew(), DBlinkStateDestroy(), DBlinkStateNew(), DBlinkTabDestroy(), DBlinkTabEnd(), DBlinkTabInit(), DBlinkTabNew(), DBlinkTabResize(), DBlspDestroy(), DBlspListDestroy(), DBlspListEnd(), DBlspListNew(), DBlspNew(), DBlspVecDestroy(), DBlspVecEnd(), DBlspVecNew(), dblVecDestroy(), dblVecEnd(), dblVecNew(), DBnew(), DBnodeDestroy(), DBnodeNew(), DBnodeVecDestroy(), DBnodeVecEnd(), DBnodeVecNew(), errorDestroy(), fillTopo(), longVecDestroy(), longVecEnd(), longVecNew(), lspRequestDestroy(), lspRequestListEnd(), lspRequestListInit(), lspRequestNew(), and myfree().

4.7.1.4 #define longListCopy(a, b) longVecCopy((LongVec∗) a,(LongVec∗) b)

Definition at line 50 of file common.h.

Referenced by DBlspCopy(), evalLS(), isValidLSPLink(), isValidRequestLink(), longListMerge(), and lsp-RequestCopy().

4.7.1.5 #define longListDestroy(a) longVecDestroy((LongVec∗) a)

Definition at line 49 of file common.h.

4.7.1.6 #define longListEnd(a) longVecEnd((LongVec∗) a)

Definition at line 48 of file common.h.

Referenced by bellmanKalaba(), computeBackup(), DBaddLSP(), DBlspDestroy(), DBlspEnd(), DBlsp-Init(), DBlspNew(), DBnodeDestroy(), DBnodeEnd(), DBnodeInit(), DBnodeNew(), fillTopo(), isValid-RequestLink(), longListMerge(), lspRequestDestroy(), lspRequestEnd(), lspRequestInit(), and lspRequest-New().

4.7.1.7 #define longListInit(a, b) longVecInit((LongVec∗) a,b)

Definition at line 47 of file common.h.

Referenced by bellmanKalaba(), computeBackup(), DBaddLSP(), DBlspInit(), DBlspNew(), DBnode-Init(), DBnodeNew(), fillTopo(), isValidRequestLink(), longListMerge(), lspRequestInit(), and lspRequest-New().

4.7.1.8 #define longListNew(a) longVecNew(a)

Definition at line 46 of file common.h.

4.7.1.9 #define longListPopBack(a, b) longVecPopBack((LongVec∗) a,b)

Definition at line 53 of file common.h.

Referenced by fillTopo().

4.7.1.10 #define longListPushBack(a, b) longVecPushBack((LongVec∗) a,b)

Definition at line 52 of file common.h.

Referenced by bellmanKalaba(), chooseReroutedLSPs(), computeBackup(), computeCost(), DBaddLink(), fillTopo(), isValidRequestLink(), longListMerge(), and updateRequest().

4.7.1.11 #define longListResize(a, b) longVecResize((LongVec∗) a,b)

Definition at line 51 of file common.h.

Referenced by longListInsert().

4.7.1.12 #define malloc(a) mymalloc(a)

Definition at line 88 of file common.h.

Referenced by avl_malloc(), and mymalloc().

4.7.1.13 #define max(a, b) (((a)>(b))?(a):(b))

Definition at line 7 of file common.h.

Referenced by bkConnectVecSet(), bkNodeVecSet(), DBaddLink(), DBlinkTabSet(), DBlspVecSet(), dblVecSet(), DBnodeVecSet(), longListInsert(), longVecSet(), and updateLS().

4.7.1.14 #define min(a, b) (((a)<(b))?(a):(b))

Definition at line 8 of file common.h.

Referenced by DBlinkTabResize(), and makeRerouteScore().

4.7.1.15 #define realloc(a, b) myrealloc(a,b)

Definition at line 89 of file common.h.

Referenced by addError(), bkConnectVecCopy(), bkConnectVecPushBack(), bkConnectVecResize(), bkNodeVecResize(), DBlinkTabResize(), DBlspListInsert(), DBlspVecResize(), dblVecCopy(), dblVecPushBack(), dblVecResize(), DBnodeVecResize(), longVecCopy(), longVecPushBack(), longVecResize(), lspRequestListResize(), and myrealloc().

4.7.2 Typedef Documentation

4.7.2.1 typedef unsigned char bool

Definition at line 5 of file common.h.

4.7.2.2 typedef struct DoubleVec_ DoubleVec

4.7.2.3 typedef LongVec LongList

Definition at line 44 of file common.h.

4.7.2.4 typedef struct LongVec_ LongVec

4.7.3 Enumeration Type Documentation

4.7.3.1 anonymous enum

Enumeration values:

FALSE

TRUE

Definition at line 4 of file common.h.

```
4 {FALSE=0,TRUE=1};
```

4.7.4 Function Documentation

4.7.4.1 int dblVecCopy (DoubleVec *, DoubleVec *)

Definition at line 529 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, realloc, DoubleVec_::size, and DoubleVec_::top.

Referenced by DBlinkStateCopy().

```
{
    double *ptr=NULL;

    if (dst == NULL || dst->cont == NULL ||
        src == NULL || src->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (dst->size < src->size)
    {
        if ((ptr=realloc(dst->cont,src->size*sizeof(double)))==NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            dst->cont=ptr;
            dst->size=src->size;
        }
    }

    memcpy(dst->cont,src->cont,src->size*sizeof(double));
    dst->top=src->top;

    return 0;
}
```

4.7.4.2 int dblVecDestroy (DoubleVec *)

Definition at line 514 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, and free.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.7.4.3 int dblVecEnd (DoubleVec ∗)

Definition at line 497 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, free, DoubleVec_::size, and DoubleVec_::top.

Referenced by DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), and DBlinkStateNew().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.7.4.4 int dblVecGet (DoubleVec ∗, long, double ∗)

Definition at line 640 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, and DoubleVec_::size.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[index];

    return 0;

}
```

4.7.4.5 int dblVecInit (DoubleVec *, long)

Definition at line 469 of file common.c.

References addError(), calloc, DoubleVec_::cont, CRITICAL, DBLVEC_INITSIZE, DoubleVec_::size, and DoubleVec_::top.

Referenced by DBlinkStateInit(), and DBlinkStateNew().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = DBLVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(double))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.7.4.6 DoubleVec* dblVecNew (long)

Definition at line 439 of file common.c.

References addError(), calloc, DoubleVec_::cont, CRITICAL, DBLVEC_INITSIZE, free, DoubleVec_::size, and DoubleVec_::top.

```
{
    DoubleVec* vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(DoubleVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = DBLVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(double))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.7.4.7 int dblVecPopBack (DoubleVec ∗, double ∗)

Definition at line 592 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, and DoubleVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[vec->top - 1];
    vec->top--;

    return 0;
}
```

4.7.4.8 int dblVecPushBack (DoubleVec ∗, double)

Definition at line 562 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, realloc, DoubleVec_::size, and DoubleVec_::top.

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if ((ptr = realloc(vec->cont, vec->size *
                           2 * sizeof(double))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        vec->size *= 2;
        vec->cont = ptr;
    }

    vec->cont[vec->top++] = val;

    return 0;
}
```

4.7.4.9 int dblVecResize (DoubleVec *, long)

Definition at line 615 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, realloc, and DoubleVec_::size.

Referenced by dblVecSet(), and updateLS().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = realloc(vec->cont, newsize*sizeof(double))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont = ptr;
    memset(ptr+ (vec->size * sizeof(double)), 0, (newsize - vec->size)*sizeof(double));
    vec->size = newsize;

    return 0;
}
```

4.7.4.10 int dblVecSet (DoubleVec *, long, double)

Definition at line 669 of file common.c.

References addError(), DoubleVec_::cont, CRITICAL, dblVecResize(), max, DoubleVec_::size, and DoubleVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        if (dblVecResize(vec,max(vec->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize double vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    vec->cont[index] = val;
    vec->top = max(vec->top, index+1);

    return 0;
}
```

4.7.4.11 int longListInsert (LongList ∗, long, long)

Definition at line 281 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longListResize, max, LongVec_::size, and LongVec_::top.

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((list->top >= list->size) || (index >= list->size))
    {
        if (longListResize(list,max(list->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize long vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (index < list->top)
    {
        memmove(list->cont+index+1,list->cont+index,(list->top-index) * sizeof(long));
    }
    list->cont[index]=val;
    list->top++;

    return 0;
}
```

4.7.4.12 int longListMerge (LongList ∗, LongList ∗, LongList ∗)

Definition at line 357 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longCompare(), longListCopy, longListEnd, longListInit, longListPushBack, and LongVec_::top.

Referenced by DBaddLSP().

```
{
    int i=0,j=0;
    LongList tmpList;

    if (la == NULL || la->cont == NULL ||
        lb == NULL || lb->cont == NULL ||
        dest == NULL || dest->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    qsort(la->cont,la->top,sizeof(long),&longCompare);
    qsort(lb->cont,lb->top,sizeof(long),&longCompare);

    if (longListInit(&tmpList,la->top+lb->top)<0)
    {
        addError(CRITICAL,"Unable to initialize temporary list of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    while ((i<la->top) || (j<lb->top))
    {
        if ((j>=lb->top) || (i<la->top && la->cont[i]<lb->cont[j]))
        {
            longListPushBack(&tmpList,la->cont[i]);
            i++;
        }
        else if ((i>=la->top) || (j<lb->top && la->cont[i]>lb->cont[j]))
        {
            longListPushBack(&tmpList,lb->cont[j]);
            j++;
        }
        else if (la->cont[i]==lb->cont[j])
        {
            longListPushBack(&tmpList,la->cont[i]);
            i++;
            j++;
        }
        else
        {
            addError(CRITICAL,"Internal error in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&tmpList);
            return -1;
        }
    }

    if (longListCopy(dest,&tmpList)<0)
    {
        addError(CRITICAL,"Unable to create resulting merged list of longs in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&tmpList);
        return -1;
    }

    longListEnd(&tmpList);

    return 0;
}
```

4.7.4.13 int longListRemove (LongList ∗, long)

Definition at line 317 of file common.c.

References addError(), LongVec_::cont, CRITICAL, LongVec_::top, and WARNING.

Referenced by DBremoveLink().

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index>=list->top)
    {
        addError(WARNING,"Removing inexistent list element in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    else
    {
        memmove(list->cont+index,list->cont+index+1,(list->top-index-1) * sizeof(long));
        list->top--;
        return 0;
    }
}
```

4.7.4.14 int longListSort (LongList ∗)

Definition at line 421 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longCompare(), and LongVec_::top.

Referenced by DBaddLink().

```
                                 {

    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    qsort(list->cont,list->top,sizeof(long),&longCompare);

    return 0;
}
```

4.7.4.15 int longVecCopy (LongVec ∗, LongVec ∗)

Definition at line 106 of file common.c.

References addError(), LongVec_::cont, CRITICAL, realloc, LongVec_::size, and LongVec_::top.

```
{
    long *ptr=NULL;

    if (dst == NULL || dst->cont == NULL ||
        src == NULL || src->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (dst->size < src->size)
    {
        if ((ptr=(long*) realloc(dst->cont,src->size*sizeof(long)))==NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            dst->cont=ptr;
            dst->size=src->size;
        }
    }

    memcpy(dst->cont,src->cont,src->size*sizeof(long));
    dst->top=src->top;

    return 0;
}
```

4.7.4.16 int longVecDestroy (LongVec ∗)

Definition at line 91 of file common.c.

References addError(), LongVec_::cont, CRITICAL, and free.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.7.4.17 int longVecEnd (LongVec ∗)

Definition at line 74 of file common.c.

References addError(), LongVec_::cont, CRITICAL, free, LongVec_::size, and LongVec_::top.

Referenced by DBaddLSP(), DBdestroy(), DBnew(), and endTopo().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.7.4.18 int longVecGet (LongVec ∗, long, long ∗)

Definition at line 216 of file common.c.

References addError(), LongVec_::cont, CRITICAL, and LongVec_::size.

Referenced by DBgetLinkDst(), and DBgetLinkSrc().

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[index];

    return 0;

}
```

4.7.4.19 int longVecInit (LongVec *, long)

Definition at line 46 of file common.c.

References addError(), calloc, LongVec_::cont, CRITICAL, LONGVEC_INITSIZE, LongVec_::size, and LongVec_::top.

Referenced by DBaddLSP(), DBnew(), and initTopo().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LONGVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.7.4.20 LongVec* longVecNew (long)

Definition at line 16 of file common.c.

References addError(), calloc, LongVec_::cont, CRITICAL, free, LONGVEC_INITSIZE, LongVec_::size, and LongVec_::top.

```
{
    LongVec* vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(LongVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LONGVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.7.4.21 int longVecPopBack (LongVec ∗, long ∗)

Definition at line 169 of file common.c.

References addError(), LongVec_::cont, CRITICAL, and LongVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    *val = vec->cont[vec->top - 1];
    vec->top--;

    return 0;
}
```

4.7.4.22 int longVecPushBack (LongVec ∗, long)

Definition at line 139 of file common.c.

References addError(), LongVec_::cont, CRITICAL, realloc, LongVec_::size, and LongVec_::top.

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if ((ptr = realloc(vec->cont, vec->size *
                           2 * sizeof(long))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        vec->size *= 2;
        vec->cont = ptr;
    }

    vec->cont[vec->top++] = val;

    return 0;
}
```

4.7.4.23 int longVecResize (LongVec ∗, long)

Definition at line 191 of file common.c.

References addError(), LongVec_::cont, CRITICAL, realloc, and LongVec_::size.

Referenced by longVecSet().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = realloc(vec->cont, newsize*sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont = ptr;
    memset(ptr+ (vec->size * sizeof(long)), 0, (newsize - vec->size)*sizeof(long));
    vec->size = newsize;

    return 0;
}
```

4.7.4.24 int longVecSet (LongVec ∗, long, long)

Definition at line 245 of file common.c.

References addError(), LongVec_::cont, CRITICAL, longVecResize(), max, LongVec_::size, and LongVec_::top.

Referenced by DBaddLink(), DBremoveLink(), and fillTopo().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        if (longVecResize(vec,max(vec->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize long vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    vec->cont[index] = val;
    vec->top = max(vec->top, index+1);

    return 0;
}
```

4.7.4.25 void∗ mycalloc (size_t, size_t)

Definition at line 751 of file common.c.

References mymalloc().

```
{
    void *ptr;

    if ((ptr=mymalloc(nmemb*sz))==NULL)
        return NULL;

    memset(ptr,0,nmemb*sz);

    return ptr;
}
```

4.7.4.26 void myfree (void ∗)

Definition at line 741 of file common.c.

References allocatedMemory, and free.

```
{
    if (*((long*)(ptr-sizeof(long))) == -1)
        fprintf(stderr, "Warning already freed\n");

    allocatedMemory-=*((long*)(ptr-sizeof(long)));
    *((long*)(ptr-sizeof(long)))=-1;
    free(ptr-sizeof(long));
}
```

4.7.4.27 void∗ mymalloc (size_t)

Definition at line 715 of file common.c.

References allocatedMemory, and malloc.

Referenced by mycalloc().

```
{
    void *ptr;

    if ((ptr=malloc(sz+sizeof(long)))==NULL)
        return NULL;

    allocatedMemory+=sz;
    *((long*)ptr)=sz;

    return (ptr+sizeof(long));
}
```

4.7.4.28 void∗ myrealloc (void ∗, size_t)

Definition at line 728 of file common.c.

References allocatedMemory, and realloc.

```
{
    void *retptr;

    if ((retptr=realloc(ptr-sizeof(long),sz+sizeof(long)))==NULL)
        return NULL;

    allocatedMemory+=sz-(*((long*)(retptr)));
    *((long*)retptr)=sz;

    return (retptr+sizeof(long));
}
```

4.7.5 Variable Documentation

4.7.5.1 long allocatedMemory

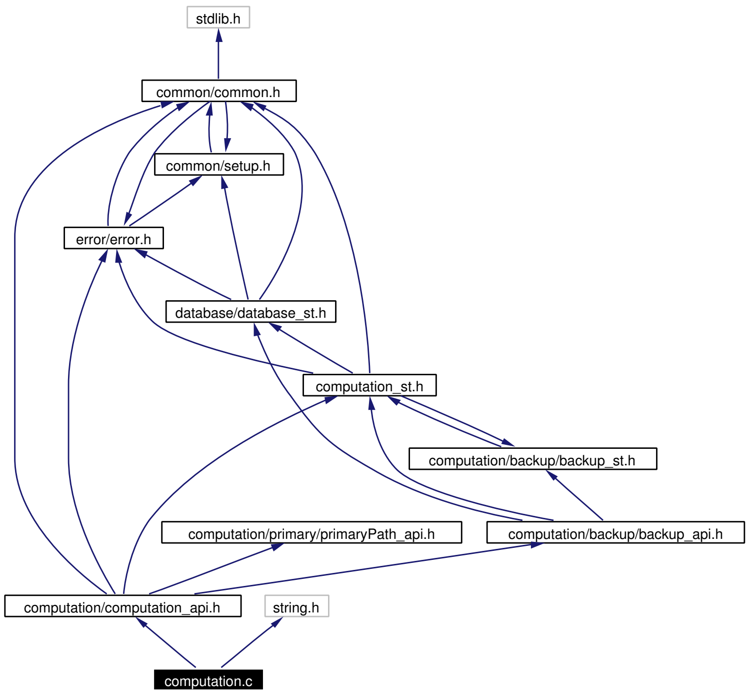
Definition at line 86 of file common.h.

Referenced by myfree(), mymalloc(), and myrealloc().

4.8 computation.c File Reference

```
#include "computation/computation_api.h"
#include <string.h>
```

Include dependency graph for computation.c:


stdlib.h
common/common.h
common/setup.h
error/error.h
database/database_st.h
computation_st.h
computation/backup/backup_st.h
computation/primary/primaryPath_api.h
computation/backup/backup_api.h
computation/computation_api.h
string.h
computation.c

Functions

- LSPRequest ∗ lspRequestNew ()
- int lspRequestInit (LSPRequest ∗req)
- int lspRequestDestroy (LSPRequest ∗req)
- int lspRequestEnd (LSPRequest ∗req)
- int lspRequestCopy (LSPRequest ∗dst, LSPRequest ∗src)
- int lspRequestListInit (LSPRequestList ∗reqList, long size)
- int lspRequestListEnd (LSPRequestList ∗reqList)
- int lspRequestListResize (LSPRequestList ∗reqList, long size)
- long lspRequestListSize (LSPRequestList ∗reqList)
- LSPRequest ∗ lspRequestListGet (LSPRequestList ∗reqList, long index)
- int lspRequestListSet (LSPRequestList ∗reqList, LSPRequest ∗req, long index)

4.8.1 Function Documentation

4.8.1.1 int lspRequestCopy (LSPRequest ∗ *dst*, LSPRequest ∗ *src*)

Definition at line 111 of file computation.c.

References addError(), ANDERROR, CRITICAL, LSPRequest_::forbidLinks, LSPRequest_::id, longListCopy, LSPRequest_::path, LSPRequest_::precedence, LSPRequest_::primID, LSPRequest_::rerouteInfo, and LSPRequest_::type.

```
{
    int ret=0;

    if (dst == NULL || src==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    dst->id=src->id;
    dst->primID=src->primID;
    dst->precedence=src->precedence;
    dst->type=src->type;
    memmove(&(dst->rerouteInfo), &(src->rerouteInfo), sizeof(LSPrerouteInfo));
    ANDERROR(ret,longListCopy(&(dst->forbidLinks),&(src->forbidLinks)));
    ANDERROR(ret,longListCopy(&(dst->path),&(src->path)));

    if (ret<0)
    {
        addError(CRITICAL,"Label switched path request copy uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.8.1.2 int lspRequestDestroy (LSPRequest * *req*)

Definition at line 80 of file computation.c.

References addError(), CRITICAL, LSPRequest_::forbidLinks, free, longListEnd, and LSPRequest_::path.

```
{
    if (req == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListEnd(&(req->forbidLinks));
    longListEnd(&(req->path));
    free(req);

    return 0;
}
```

4.8.1.3 int lspRequestEnd (LSPRequest * *req*)

Definition at line 96 of file computation.c.

References addError(), CRITICAL, LSPRequest_::forbidLinks, longListEnd, and LSPRequest_::path.

Referenced by computeCost(), isValidLSPLink(), lspRequestListEnd(), lspRequestListInit(), and lspRequestListResize().

```
{
    if (req == NULL)
    {
         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
         return -1;
     }

     longListEnd(&(req->forbidLinks));
     longListEnd(&(req->path));

     return 0;
}
```

4.8.1.4 int lspRequestInit (LSPRequest ∗ *req*)

Definition at line 47 of file computation.c.

References addError(), LSPRequest_::bw, CRITICAL, LSPRequest_::forbidLinks, LSPrerouteInfo_::id, longListEnd, longListInit, NB_OA, LSPRequest_::path, and LSPRequest_::rerouteInfo.

Referenced by computeCost(), isValidLSPLink(), lspRequestListInit(), and lspRequestListResize().

```
{
    if (req == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(req,0,sizeof(LSPRequest));

    if (longListInit(&(req->forbidLinks),-1)<0)
    {
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(req->path),-1)<0)
    {
        longListEnd(&(req->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(req->bw, 0, NB_OA * sizeof(double));

    req->rerouteInfo.id = -1;

    return 0;
}
```

4.8.1.5 int lspRequestListEnd (LSPRequestList ∗ *reqList*)

Definition at line 184 of file computation.c.

References addError(), LSPRequestList_::cont, CRITICAL, free, lspRequestEnd(), and LSPRequestList_::size.

```
{
    long i;

    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<reqList->size; i++)
    {
        lspRequestEnd(&(reqList->cont[i]));
    }

    free(reqList->cont);

    return 0;
}
```

4.8.1.6 LSPRequest∗ lspRequestListGet (LSPRequestList ∗ *reqList*, long *index*)

Definition at line 269 of file computation.c.

References addError(), LSPRequestList_::cont, CRITICAL, and LSPRequestList_::size.

Referenced by computeBackup().

```
{
    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (index < 0 || index >= reqList->size)
    {
        addError(CRITICAL,"Out of bound index in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    return &(reqList->cont[index]);
}
```

4.8.1.7 int lspRequestListInit (LSPRequestList ∗ *reqList*, long *size*)

Definition at line 143 of file computation.c.

References addError(), calloc, LSPRequestList_::cont, CRITICAL, free, LSPREQLIST_INITSIZE, lspRequestEnd(), lspRequestInit(), and LSPRequestList_::size.

```
{
    long i;

    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    size = (size<=0?LSPREQLIST_INITSIZE:size);

    if ((reqList->cont = calloc(size, sizeof(LSPRequest))) == NULL)
    {
        addError(CRITICAL,"Impossible to allocate memory for LSPRequestList in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<size; i++)
    {
        if (lspRequestInit(&(reqList->cont[i])) < 0)
        {
            addError(CRITICAL,"Error while initialisation of LSPRequest in %s at line %d",
                     __FILE__,__LINE__);
            // clean up
            for (i--;i>=0;i--)
            {
                lspRequestEnd(&(reqList->cont[i]));
            }
            free(reqList->cont);
            return -1;
        }
    }

    reqList->size = size;

    return 0;
}
```

4.8.1.8 int lspRequestListResize (LSPRequestList * *reqList*, long *size*)

Definition at line 205 of file computation.c.

References addError(), LSPRequestList_::cont, CRITICAL, lspRequestEnd(), lspRequestInit(), realloc, and LSPRequestList_::size.

Referenced by computeBackup().

```
{
    long i;

    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (reqList->size < size)
    {
        LSPRequest* ptr;

        if ((ptr = realloc(reqList->cont, size*sizeof(LSPRequest))) == NULL)
        {
            addError(CRITICAL,"Impossible to allocate memory for LSPRequestList in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        reqList->cont = ptr;

        for (i=reqList->size; i<size; i++)
        {
            if (lspRequestInit(&(reqList->cont[i])) < 0)
            {
                addError(CRITICAL,"Error while initialisation of LSPRequest in %s at line %d",
                         __FILE__,__LINE__);
                // clean up
                for (i--;i>=reqList->size;i--)
                {
                    lspRequestEnd(&(reqList->cont[i]));
                }
                return -1;
            }
        }
        reqList->size = size;
    }
    else
    {
        for (i=reqList->size-1; i>=size; i--)
        {
            lspRequestEnd(&(reqList->cont[i]));
        }
        reqList->size = size;
    }

    return 0;
}
```

4.8.1.9 int lspRequestListSet (LSPRequestList ∗ *reqList*, LSPRequest ∗ *req*, long *index*)

Definition at line 288 of file computation.c.

References addError(), and CRITICAL.

```
{
    addError(CRITICAL,"Trying to call an undefined function in %s at line %d",
             __FILE__,__LINE__);

    return 0;
}
```

4.8.1.10 long lspRequestListSize (LSPRequestList ∗ *reqList*)

Definition at line 257 of file computation.c.

References addError(), CRITICAL, and LSPRequestList_::size.

```
{
    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return reqList->size;
}
```

4.8.1.11 LSPRequest∗ lspRequestNew ()

Definition at line 12 of file computation.c.

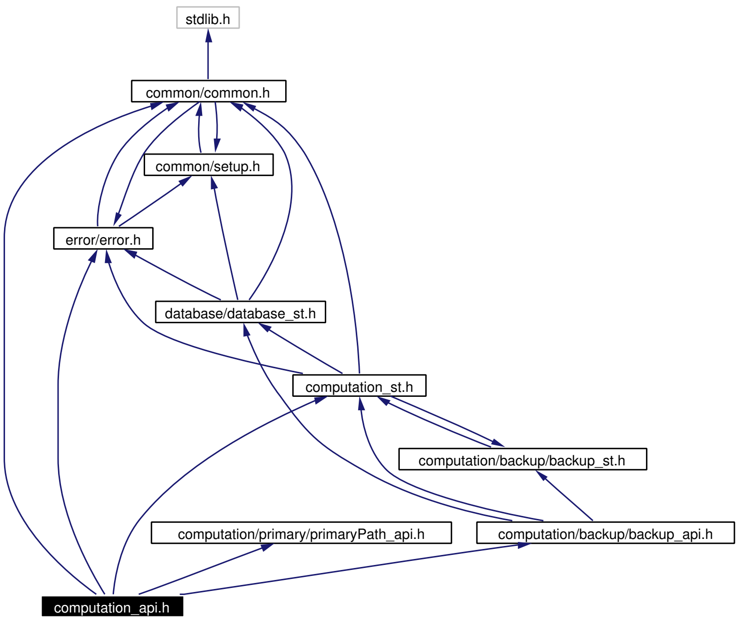
References addError(), LSPRequest_::bw, calloc, CRITICAL, LSPRequest_::forbidLinks, free, LSPrerouteInfo_::id, longListEnd, longListInit, NB_OA, LSPRequest_::path, and LSPRequest_::rerouteInfo.

```
{
    LSPRequest* req;

    if ((req=calloc(1,sizeof(LSPRequest)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(req->forbidLinks),-1)<0)
    {
        free(req);
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(req->path),-1)<0)
    {
        longListEnd(&(req->forbidLinks));
        free(req);
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    memset(req->bw, 0, NB_OA * sizeof(double));

    req->rerouteInfo.id = -1;

    return req;
}
```

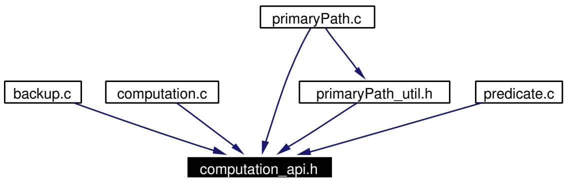
4.9 computation_api.h File Reference

```
#include "common/common.h"
#include "error/error.h"
#include "computation_st.h"
#include "computation/primary/primaryPath_api.h"
#include "computation/backup/backup_api.h"
```

Include dependency graph for computation_api.h:



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



Functions

- LSPRequest ∗ lspRequestNew ()
- int lspRequestInit (LSPRequest ∗)
- int lspRequestDestroy (LSPRequest ∗)
- int lspRequestEnd (LSPRequest ∗)
- int lspRequestCopy (LSPRequest ∗, LSPRequest ∗)
- int lspRequestListInit (LSPRequestList ∗, long)

- int lspRequestListEnd (LSPRequestList ∗)
- int lspRequestListResize (LSPRequestList ∗, long)
- long lspRequestListsize (LSPRequestList ∗)
- LSPRequest ∗ lspRequestListGet (LSPRequestList ∗, long)
- int lspRequestListSet (LSPRequestList ∗, LSPRequest ∗, long)

4.9.1 Function Documentation

4.9.1.1 int lspRequestCopy (LSPRequest ∗, LSPRequest ∗)

Definition at line 111 of file computation.c.

References addError(), ANDERROR, CRITICAL, LSPRequest_::forbidLinks, LSPRequest_::id, longListCopy, LSPRequest_::path, LSPRequest_::precedence, LSPRequest_::primID, LSPRequest_::rerouteInfo, and LSPRequest_::type.

```
112 {
113     int ret=0;
114 
115     if (dst == NULL || src==NULL)
116     {
117         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
118                  __FILE__,__LINE__);
119         return -1;
120     }
121 
122     dst->id=src->id;
123     dst->primID=src->primID;
124     dst->precedence=src->precedence;
125     dst->type=src->type;
126     memmove(&(dst->rerouteInfo), &(src->rerouteInfo), sizeof(LSPrerouteInfo));
127     ANDERROR(ret,longListCopy(&(dst->forbidLinks),&(src->forbidLinks)));
128     ANDERROR(ret,longListCopy(&(dst->path),&(src->path)));
129 
130     if (ret<0)
131     {
132         addError(CRITICAL,"Label switched path request copy uncomplete in %s at line %d",
133                  __FILE__,__LINE__);
134     }
135 
136     return ret;
137 }
```

4.9.1.2 int lspRequestDestroy (LSPRequest ∗)

Definition at line 80 of file computation.c.

References addError(), CRITICAL, LSPRequest_::forbidLinks, free, longListEnd, and LSPRequest_::path.

```
81 {
82     if (req == NULL)
83     {
84         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
85                  __FILE__,__LINE__);
86         return -1;
87     }
88 
89     longListEnd(&(req->forbidLinks));
90     longListEnd(&(req->path));
```

```
    free(req);

    return 0;
}
```

4.9.1.3 int lspRequestEnd (LSPRequest ∗)

Definition at line 96 of file computation.c.

References addError(), CRITICAL, LSPRequest_::forbidLinks, longListEnd, and LSPRequest_::path.

Referenced by computeCost(), isValidLSPLink(), lspRequestListEnd(), lspRequestListInit(), and lsp-RequestListResize().

```
{
    if (req == NULL)
    {
         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
         return -1;
     }

     longListEnd(&(req->forbidLinks));
     longListEnd(&(req->path));

     return 0;
}
```

4.9.1.4 int lspRequestInit (LSPRequest ∗)

Definition at line 47 of file computation.c.

References addError(), LSPRequest_::bw, CRITICAL, LSPRequest_::forbidLinks, LSPrerouteInfo_::id, longListEnd, longListInit, NB_OA, LSPRequest_::path, and LSPRequest_::rerouteInfo.

Referenced by computeCost(), isValidLSPLink(), lspRequestListInit(), and lspRequestListResize().

```
{
    if (req == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(req,0,sizeof(LSPRequest));

    if (longListInit(&(req->forbidLinks),-1)<0)
    {
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(req->path),-1)<0)
    {
        longListEnd(&(req->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(req->bw, 0, NB_OA * sizeof(double));

    req->rerouteInfo.id = -1;

    return 0;
}
```

4.9.1.5 int lspRequestListEnd (LSPRequestList ∗)

Definition at line 184 of file computation.c.

References addError(), LSPRequestList_::cont, CRITICAL, free, lspRequestEnd(), and LSPRequestList_-::size.

```
{
    long i;

    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<reqList->size; i++)
    {
        lspRequestEnd(&(reqList->cont[i]));
    }

    free(reqList->cont);

    return 0;
}
```

4.9.1.6 LSPRequest∗ lspRequestListGet (LSPRequestList ∗, long)

Definition at line 269 of file computation.c.

References addError(), LSPRequestList_::cont, CRITICAL, and LSPRequestList_::size.

Referenced by computeBackup().

```
{
    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (index < 0 || index >= reqList->size)
    {
        addError(CRITICAL,"Out of bound index in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    return &(reqList->cont[index]);
}
```

4.9.1.7 int lspRequestListInit (LSPRequestList ∗, long)

Definition at line 143 of file computation.c.

References addError(), calloc, LSPRequestList_::cont, CRITICAL, free, LSPREQLIST_INITSIZE, lspRequestEnd(), lspRequestInit(), and LSPRequestList_::size.

```
{
    long i;

    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    size = (size<=0?LSPREQLIST_INITSIZE:size);

    if ((reqList->cont = calloc(size, sizeof(LSPRequest))) == NULL)
    {
        addError(CRITICAL,"Impossible to allocate memory for LSPRequestList in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<size; i++)
    {
        if (lspRequestInit(&(reqList->cont[i])) < 0)
        {
            addError(CRITICAL,"Error while initialisation of LSPRequest in %s at line %d",
                 __FILE__,__LINE__);
            // clean up
            for (i--;i>=0;i--)
            {
                lspRequestEnd(&(reqList->cont[i]));
            }
            free(reqList->cont);
            return -1;
        }
    }

    reqList->size = size;

    return 0;
}
```

4.9.1.8 int lspRequestListResize (LSPRequestList ∗, long)

Definition at line 205 of file computation.c.

References addError(), LSPRequestList_::cont, CRITICAL, lspRequestEnd(), lspRequestInit(), realloc, and LSPRequestList_::size.

Referenced by computeBackup().

```
{
    long i;

    if (reqList == NULL)
    {
        addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
                 __FILE__,__LINE__);
```

```
            return -1;
        }

        if (reqList->size < size)
        {
            LSPRequest* ptr;

            if ((ptr = realloc(reqList->cont, size*sizeof(LSPRequest))) == NULL)
            {
                addError(CRITICAL,"Impossible to allocate memory for LSPRequestList in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }

            reqList->cont = ptr;

            for (i=reqList->size; i<size; i++)
            {
                if (lspRequestInit(&(reqList->cont[i])) < 0)
                {
                    addError(CRITICAL,"Error while initialisation of LSPRequest in %s at line %d",
                             __FILE__,__LINE__);
                    // clean up
                    for (i--;i>=reqList->size;i--)
                    {
                        lspRequestEnd(&(reqList->cont[i]));
                    }
                    return -1;
                }
            }
            reqList->size = size;
        }
        else
        {
            for (i=reqList->size-1; i>=size; i--)
            {
                lspRequestEnd(&(reqList->cont[i]));
            }
            reqList->size = size;
        }

        return 0;
    }
```

4.9.1.9 int lspRequestListSet (LSPRequestList *, LSPRequest *, long)

Definition at line 288 of file computation.c.

References addError(), and CRITICAL.

```
    {
        addError(CRITICAL,"Trying to call an undefined function in %s at line %d",
                 __FILE__,__LINE__);

        return 0;
    }
```

4.9.1.10 long lspRequestListsize (LSPRequestList *)

4.9.1.11 LSPRequest* lspRequestNew ()

Definition at line 12 of file computation.c.

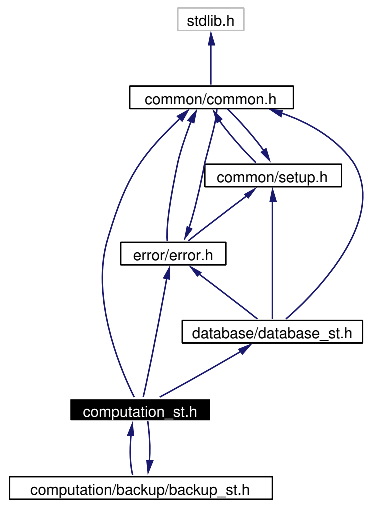
References addError(), LSPRequest_::bw, calloc, CRITICAL, LSPRequest_::forbidLinks, free, LSPrerouteInfo_::id, longListEnd, longListInit, NB_OA, LSPRequest_::path, and LSPRequest_::rerouteInfo.

```
{
    LSPRequest* req;

    if ((req=calloc(1,sizeof(LSPRequest)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(req->forbidLinks),-1)<0)
    {
        free(req);
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(req->path),-1)<0)
    {
        longListEnd(&(req->forbidLinks));
        free(req);
        addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    memset(req->bw, 0, NB_OA * sizeof(double));

    req->rerouteInfo.id = -1;

    return req;
}
```

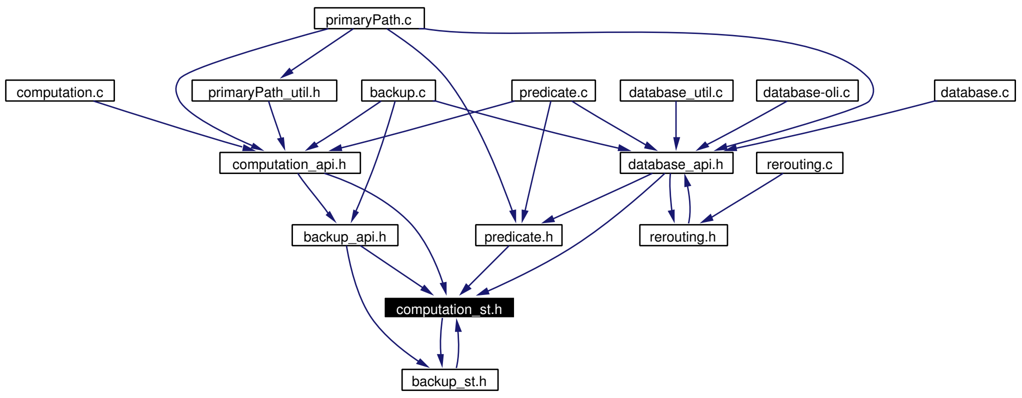
4.10 computation_st.h File Reference

```
#include "common/common.h"
#include "error/error.h"
#include "database/database_st.h"
#include "computation/backup/backup_st.h"
```

Include dependency graph for computation_st.h:



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



Data Structures

- struct LSPRequest_

  LSP Request Structure.

- struct LSPRequestList_
- struct LSPrerouteInfo_

Rerouting Information structure.

Typedefs

- typedef LSPrerouteInfo_ LSPrerouteInfo

  Rerouting Information structure.

- typedef LSPRequest_ LSPRequest

  LSP Request Structure.

- typedef LSPRequestList_ LSPRequestList

4.10.1 Typedef Documentation

4.10.1.1 typedef struct LSPRequest_ LSPRequest

LSP Request Structure.

Label Switched Path request representation, used by computePrimaryPath

4.10.1.2 typedef struct LSPRequestList_ LSPRequestList

4.10.1.3 typedef struct LSPrerouteInfo_ LSPrerouteInfo

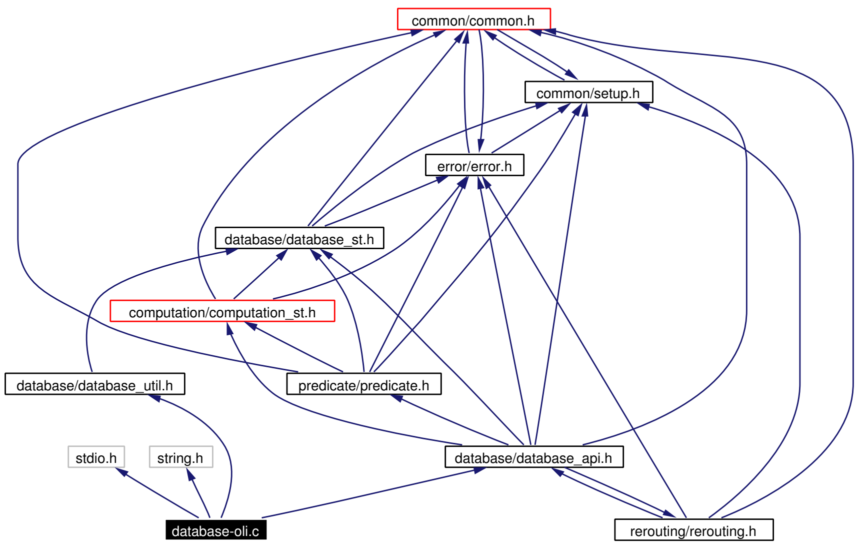
Rerouting Information structure.

Used to support soft preemption. When a LSP is preempted, we have two choices. 1. Tear down this LSP immediately, this is hard preemption. 2. Notice the entity responsible for this LSP (e.g. the ingress in a decentralized mode) so that it can reestablish another LSP before the preempted one is being torn down. This is soft preemption. When soft preemption is used, when the computation of the new LSP (meant for replacing the soon preempted one) occurs, the computation algorithm must take into account the fact that the ressources of the preempted one can be used. But it is also interesting to take into account the link where the preemption occured, because it's certainly a link that must no more be used. In a decentralized approach, there's a good probability that the topology representation that the ingress has is not up-to-date when computing the rerouting. So, this is at least an interesting information to give to the computation algorithm.

4.11 database-oli.c File Reference

```
#include "database/database_api.h"
#include "database/database_util.h"
#include <stdio.h>
#include <string.h>
```

Include dependency graph for database-oli.c:



Typedefs

- typedef enum operation_ operation

Enumerations

- enum operation_ { SETUP, REMOVE }

Functions

- DBLabelSwitchedPath ∗ DBlspNew ()
- int DBlspInit (DBLabelSwitchedPath ∗lsp)
- int DBlspDestroy (DBLabelSwitchedPath ∗lsp)
- int DBlspEnd (DBLabelSwitchedPath ∗lsp)
- int DBlspCopy (DBLabelSwitchedPath ∗dst, DBLabelSwitchedPath ∗src)
- DBLSPList ∗ DBlspListNew (long size)
- int DBlspListInit (DBLSPList ∗list, long size)
- int DBlspListDestroy (DBLSPList ∗list)
- int DBlspListEnd (DBLSPList ∗list)
- int DBlspListInsert (DBLSPList ∗list, DBLabelSwitchedPath ∗lsp)
- int DBlspCompare (const DBLabelSwitchedPath ∗LSPa, const DBLabelSwitchedPath ∗LSPb)

- int DBlspListRemove (DBLSPList ∗list, DBLabelSwitchedPath ∗lsp)
- DBLinkState ∗ DBlinkStateNew ()
- int DBlinkStateInit (DBLinkState ∗ls)
- int DBlinkStateDestroy (DBLinkState ∗ls)
- int DBlinkStateEnd (DBLinkState ∗ls)
- int DBlinkStateCopy (DBLinkState ∗dst, DBLinkState ∗src)
- int computeRBW (DataBase ∗dataBase, double rbw[NB_OA][NB_PREEMPTION], double pbw[NB_OA][NB_PREEMPTION], DoubleVec bbw[NB_OA][NB_PREEMPTION], DoubleVec fbw[NB_OA][NB_PREEMPTION])
- int updateLS (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗, operation)
- int evalLS (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS, DBLinkState ∗oldLS, LSPRequest ∗req, operation op)
- int DBevalLSOnSetup (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS, DBLinkState ∗oldLS, LSPRequest ∗req)
- int DBevalLSOnRemove (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS, DBLinkState ∗oldLS, LSPRequest ∗req)
- int DBupdateLSOnSetup (DataBase ∗dataBase, long src, long dst, DBLinkState ∗ls, DBLabelSwitchedPath ∗lsp)
- int DBupdateLSOnRemove (DataBase ∗dataBase, long src, long dst, DBLinkState ∗ls, DBLabelSwitchedPath ∗lsp)
- DataBase ∗ DBnew (long ID)
- int DBdestroy (DataBase ∗dataBase)
- long DBgetID (DataBase ∗dataBase)
- long DBgetLinkID (DataBase ∗dataBase, long src, long dst)
- long DBgetLinkSrc (DataBase ∗dataBase, long id)
- long DBgetLinkDst (DataBase ∗dataBase, long id)
- long DBgetNbNodes (DataBase ∗dataBase)
- long DBgetMaxNodeID (DataBase ∗dataBase)
- int DBaddNode (DataBase ∗dataBase, long id)
- int DBremoveNode (DataBase ∗dataBase, long id)
- long DBgetNbLinks (DataBase ∗dataBase)
- int DBaddLink (DataBase ∗dataBase, long id, long src, long dst, DBLinkState ∗initLinkState)
- int DBremoveLink (DataBase ∗dataBase, long src, long dst)
- int DBaddLSP (DataBase ∗dataBase, DBLabelSwitchedPath ∗lsp, LongList ∗preemptList)
- int DBremoveLSP (DataBase ∗dataBase, long id)
- DBLabelSwitchedPath ∗ DBgetLSP (DataBase ∗dataBase, long id)
- DBLSPList ∗ DBgetLinkLSPs (DataBase ∗dataBase, long src, long dst)
- DBLinkState ∗ DBgetLinkState (DataBase ∗dataBase, long src, long dst)
- int DBsetLinkState (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS)
- LongList ∗ DBgetNodeInNeighb (DataBase ∗dataBase, long id)
- LongList ∗ DBgetNodeOutNeighb (DataBase ∗dataBase, long id)
- void DBprintDB (DataBase ∗db)

4.11.1 Typedef Documentation

4.11.1.1 typedef enum operation_ operation

4.11.2 Enumeration Type Documentation

4.11.2.1 enum operation_

Enumeration values:

SETUP

REMOVE

Definition at line 834 of file database-oli.c.

```
834 { SETUP, REMOVE} operation;
```

4.11.3 Function Documentation

4.11.3.1 int computeRBW (DataBase * *dataBase*, double *rbw*[NB_OA][NB_PREEMPTION], double *pbw*[NB_OA][NB_PREEMPTION], DoubleVec *bbw*[NB_OA][NB_PREEMPTION], DoubleVec *fbw*[NB_OA][NB_PREEMPTION])

Definition at line 694 of file database-oli.c.

References addError(), CRITICAL, NB_OA, and NB_PREEMPTION.

```
696 {
697 #if defined LINUX && defined TIME1
698     struct timezone tz;
699     struct timeval  t1,t2;
700 #endif
701     int nbLink = 0, seenLinks;
702     int nbNode = 0, seenNodes;
703     int i,oa,p;
704     DoubleVec* gbw;
705     double totBbw = 0;
706     double totFbw = 0;
707     double m, oldM;
708
709     if ((rbw==NULL) || (pbw==NULL) || (bbw==NULL) || (fbw==NULL))
710     {
711         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
712                  __FILE__,__LINE__);
713         return -1;
714     }
715
716 #if defined LINUX && defined TIME1
717     gettimeofday(&t1, &tz);
718 #endif
719
720 /*
721     nbLink = dataBase->linkSrcVec.top;
722     nbNode = dataBase->nodeVec.top;
723
724     if ((gbw = calloc(nbLink + nbNode, sizeof(DoubleVec))) == NULL)
725     {
726         addError(CRITICAL,"Cannot allocate GBW in %s at line %d",
727                  __FILE__,__LINE__);
728         return -1;
729     }
730
731     for (i=0; i<nbLink + nbNode; ++i)
732         dblVecInit(&(gbw[i]), NB_PREEMPTION);
733
734     for (oa=0; oa<NB_OA; ++oa)
735     {
736         seenLinks = 0;
737
738         // phase 1a (links)
739         for (i=0; seenLinks<dataBase->nbLinks; ++i)
740         {
741             if (dataBase->linkSrcVec.cont[i] == 0)
742                 continue;
```

```
            else
                seenLinks++;

            totBbw = 0;
            totFbw = 0;

            for(p=0; p<NB_PREEMPTION; ++p)
            {
                if (i < bbw[oa][p].size)
                    totBbw += bbw[oa][p].cont[i];
                if (i < fbw[oa][p].size)
                    totFbw += fbw[oa][p].cont[i];

                gbw[i].cont[p] = max(0, totBbw - totFbw);
            }
        }

        seenNodes = 0;

        // phase 1b (nodes or any set of links)
        for (i=0; seenNodes<dataBase->nbNodes; ++i)
        {
            if (dataBase->nodeVec.cont[i] == NULL)
                continue;
            else
                seenNodes++;

            totBbw = 0;
            totFbw = 0;

            for(p=0; p<NB_PREEMPTION; ++p)
            {
                LongList* lst;
                if ((lst = DBgetNodeInNeighb(dataBase, i)) != NULL)
                {
                    int l;
                    for (l=0; l<lst->top; ++l)
                    {
                        int lnkID = DBgetLinkID(dataBase, lst->cont[l], i);
                        if (lnkID < bbw[oa][p].size)
                            totBbw += bbw[oa][p].cont[lnkID];
                        if (lnkID < fbw[oa][p].size)
                            totFbw += fbw[oa][p].cont[lnkID];
                    }
                }

                gbw[i + nbLink].cont[p] = max(0, totBbw - totFbw);
            }
        }

        // phase 2
        oldM = 0;

        for (p=0; p<NB_PREEMPTION; ++p)
        {
            m = 0;
            for (i=0; i<nbLink + nbNode; ++i)
            {
                if (gbw[i].cont[p] > m)
                {
                    m = gbw[i].cont[p];
                }
            }

            rbw[oa][p] = pbw[oa][p] + m - oldM;
            oldM = m;
        }
```

```
    }

    for (i=0; i<nbLink + nbNode; ++i)
        dblVecEnd(&(gbw[i]));

    free(gbw);
*/

    for (oa=0; oa<NB_OA; ++oa)
        for (p=0; p<NB_PREEMPTION; ++p)
        {
            rbw[oa][p] = pbw[oa][p];
        }

#if defined LINUX && defined TIME1
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time to compute rbw : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif

    return 0;

}
```

4.11.3.2 int DBaddLink (DataBase ∗ *dataBase*, long *id*, long *src*, long *dst*, DBLinkState ∗ *initLinkState*)

Definition at line 1555 of file database-oli.c.

```
{
    DBLink* link=NULL;
    int ret=0;

    if (dataBase == NULL || initLinkState==NULL
        || id <0 || src<0 || dst<0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (((id<dataBase->linkSrcVec.size) && (dataBase->linkSrcVec.cont[id]>0))
        ||
        ((id<dataBase->linkDstVec.size) && (dataBase->linkDstVec.cont[id]>0)))
    {
        addError(CRITICAL,"Trying to add a link with a reserved ID (ID=%ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return -1;
    }

    if ((link = DBlinkNew()) == NULL)
    {
        addError(CRITICAL,"Unable to create link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    link->id=id;

    if (DBlinkStateCopy(&(link->state), initLinkState))
    {
        addError(CRITICAL,"Unable to create link in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    if ((DBnodeVecGet(&(dataBase->nodeVec),src) == NULL) ||
        (DBnodeVecGet(&(dataBase->nodeVec),dst) == NULL))
    {
        addError(CRITICAL,"Source or destination node doesn’t exist in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    if (DBlinkTabSet(&(dataBase->linkTab),link,src,dst)<0)
    {
        addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
    ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[dst]->inNeighb),src));

    ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[src]->outNeighb)));
    ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[dst]->inNeighb)));

    ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,src+1));
    ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,dst+1));

    // Maximum non-null element
    dataBase->linkSrcVec.top = max(dataBase->linkSrcVec.top, id+1);
    dataBase->linkDstVec.top = dataBase->linkSrcVec.top;

    if (ret<0)
    {
        addError(CRITICAL,"Link addition uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks++;

    return ret;
}
```

4.11.3.3 int DBaddLSP (DataBase ∗ *dataBase*, DBLabelSwitchedPath ∗ *lsp*, LongList ∗ *preemptList*)

Definition at line 1679 of file database-oli.c.

```
{
    DBLabelSwitchedPath *newLSP, *contentLSP=NULL;
    DBLSPList *lspList;
    int i,ret=0;
    DBLink *lnk=NULL;
    LongVec isProcessed;
    double rerouteGain[NB_OA];
    bool allowLSP=TRUE;
#if defined SIMULATOR
    LongList idList;
#elif defined AGENT
    int j;
    bool inPath=FALSE;
#endif

```

```
#if defined LINUX && defined TIME2
    struct timezone tz;
    struct timeval  t1,t2;
#endif

    if (dataBase == NULL || lsp==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

#if defined LINUX && defined TIME2
    gettimeofday(&t1, &tz);
#endif

    if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
    {
        addError(CRITICAL,"Unable to initialize vector of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(rerouteGain,0,NB_OA*sizeof(double));

    // Check if establishment is possible
#if defined SIMULATOR
    if (longListInit(&(idList),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize list of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    for (i=0;(i<lsp->path.top-1) && allowLSP;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
                                             &(lnk->state),lsp,rerouteGain);
        if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        idList.top=0;
        if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
        {
            addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        if (longListMerge(&(idList),preemptList,preemptList)<0)
        {
            addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        isProcessed.cont[lnk->id] = 1;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;(i<lsp->primPath.top-1) && allowLSP;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[i],lsp->primPath.cor
                                                 &(lnk->state),lsp,rerouteGain);
                if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==
                {
                    addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at li
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                idList.top=0;
                if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0
                {
                    addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                if (longListMerge(&(idList),preemptList,preemptList)<0)
                {
                    addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
    longListEnd(&(idList));
#elif defined AGENT
    for (i=0;(i<lsp->path.top-1) && (lsp->path.cont[i]!=dataBase->id);i++);

    if (i<lsp->path.top-1)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
                                             &(lnk->state),lsp,rerouteGain);
        if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
                     __FILE__,__LINE__);
            longVecEnd(&(isProcessed));
        }
        if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)<0)
        {
            addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                     __FILE__,__LINE__);
            longVecEnd(&(isProcessed));
            return -1;
        }
        isProcessed.cont[lnk->id] = 1;
        inPath=TRUE;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (j=0;(j<lsp->primPath.top-1) && (lsp->primPath.cont[j]!=dataBase->id);j++);

        if (j<lsp->primPath.top-1)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[j],
                             lsp->primPath.cont[j+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[j],lsp->primPath.cor
                                                    &(lnk->state),lsp,rerouteGain);
                if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==
                {
                    addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at li
                             __FILE__,__LINE__);
                    longVecEnd(&(isProcessed));
                }
                if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)
                {
                    addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                             __FILE__,__LINE__);
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                isProcessed.cont[lnk->id] = 1;
            }
            inPath=TRUE;
        }
    }
    if (!inPath)
    {
        addError(CRITICAL,"Agent not concerned by this LSP in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    if (!allowLSP)
    {
        addError(CRITICAL,"LSP refused by the predicate in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }


    if ((newLSP=DBlspNew())==NULL)
    {
        addError(CRITICAL,"Unable to create LSP in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (DBlspCopy(newLSP,lsp)<0)
    {
        addError(CRITICAL,"Unable to create a valid LSP copy in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(newLSP);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (DBlspVecSet(&(dataBase->lspVec),newLSP,newLSP->id)<0)
    {
        addError(CRITICAL,"Unable to insert LSP in the general LSP container in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(newLSP);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (newLSP->noContentionId>=0)
    {
        if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),newLSP->noContentionId))==NULL)
        {
            addError(WARNING,"Unable to get no contention LSP in %s at line %d",
                     __FILE__,__LINE__);
            newLSP->noContentionId=-1;
            // not critical enough to abort
        }
        else
        {
            contentLSP->noContentionId=newLSP->id;
        }
    }

    for (i=0;i<isProcessed.size;i++)
    {
        isProcessed.cont[i]=0;
    }


#if defined SIMULATOR
    // Add the LSP to each link list and update all the linkstates (only once !!!!!)
    for (i=0;i<newLSP->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
                         newLSP->path.cont[i+1]);
        ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
        ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->path.cont[i],
                                       newLSP->path.cont[i+1], &(lnk->state), newLSP));
        isProcessed.cont[lnk->id] = 1;
    }
    if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
    {
        for (i=0;i<newLSP->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
                             newLSP->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->primPath.cont[i],
                                               newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
#elif defined AGENT
    // Add the LSP to the link attached to the agent and update the linkstate
    for (i=0;i<newLSP->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
                         newLSP->path.cont[i+1]);
        ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));

        if (newLSP->path.cont[i] == dataBase->id)
        {
            ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->path.cont[i],
                                           newLSP->path.cont[i+1], &(lnk->state), newLSP));
            isProcessed.cont[lnk->id] = 1;
        }
    }
    if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
    {
        for (i=0;i<newLSP->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
                             newLSP->primPath.cont[i+1]);

            if (newLSP->primPath.cont[i] == dataBase->id)
            {
                if (isProcessed.cont[lnk->id] == 0)
                {
                    ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->primPath.cont[i],
                                                   newLSP->primPath.cont[i+1], &(lnk->state), newLSP)
                }
                break;
            }
        }
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    longVecEnd(&(isProcessed));

#if defined LINUX && defined TIME2
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time to add a new LSP : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif


    if (ret<0)
    {
        addError(CRITICAL,"LSP addition uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.11.3.4 int DBaddNode (DataBase ∗ *dataBase*, long *id*)

Definition at line 1466 of file database-oli.c.

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((node=DBnodeNew()) == NULL)
    {
        addError(CRITICAL,"Unable to create node in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    node->id=id;

    if (DBnodeVecSet(&(dataBase->nodeVec),node,id) < 0)
    {
        addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %
                 __FILE__,__LINE__);
        DBnodeDestroy(node);
        return -1;
    }

    dataBase->nbNodes++;

    return 0;
}
```

4.11.3.5 int DBdestroy (DataBase ∗ *dataBase*)

Definition at line 1349 of file database-oli.c.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBnodeVecEnd(&(dataBase->nodeVec));
    DBlspVecEnd(&(dataBase->lspVec));
    DBlinkTabEnd(&(dataBase->linkTab));
    longVecEnd(&(dataBase->linkSrcVec));
    longVecEnd(&(dataBase->linkDstVec));

    free(dataBase);

    return 0;
}
```

4.11.3.6 int DBevalLSOnRemove (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *newLS*, DBLinkState ∗ *oldLS*, LSPRequest ∗ *req*)

Definition at line 1259 of file database-oli.c.

```
{
    return evalLS(dataBase, src, dst, newLS, oldLS, req, REMOVE);
}
```

4.11.3.7 int DBevalLSOnSetup (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *newLS*, DBLinkState ∗ *oldLS*, LSPRequest ∗ *req*)

Definition at line 1253 of file database-oli.c.

```
{
    return evalLS(dataBase, src, dst, newLS, oldLS, req, SETUP);
}
```

4.11.3.8 long DBgetID (DataBase ∗ *dataBase*)

Definition at line 1369 of file database-oli.c.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->id;
}
```

4.11.3.9 long DBgetLinkDst (DataBase ∗ *dataBase*, long *id*)

Definition at line 1421 of file database-oli.c.

```
{
    long ret;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecGet(&(dataBase->linkDstVec),id,&ret)<0)
    {
        addError(CRITICAL,"Inexistent link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return (ret-1);
}
```

4.11.3.10 long DBgetLinkID (DataBase ∗ *dataBase*, long *src*, long *dst*)

Definition at line 1381 of file database-oli.c.

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL || src < 0 || dst < 0)
    {
        addError(CRITICAL,"Bad argument (NULL or negative value) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst))==NULL)
    {
        return -1;
    }

    return lnk->id;
}
```

4.11.3.11 DBLSPList∗ DBgetLinkLSPs (DataBase ∗ *dataBase*, long *src*, long *dst*)

Definition at line 2138 of file database-oli.c.

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return NULL;
    }

    return &(lnk->lspList);
}
```

4.11.3.12 long DBgetLinkSrc (DataBase ∗ *dataBase*, long *id*)

Definition at line 1400 of file database-oli.c.

```
{
    long ret;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecGet(&(dataBase->linkSrcVec),id,&ret)<0)
    {
        addError(CRITICAL,"Inexistent link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return (ret-1);
}
```

4.11.3.13 DBLinkState∗ DBgetLinkState (DataBase ∗ *dataBase*, long *src*, long *dst*)

Definition at line 2159 of file database-oli.c.

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return NULL;
    }

    return &(lnk->state);
}
```

4.11.3.14 DBLabelSwitchedPath∗ DBgetLSP (DataBase ∗ *dataBase*, long *id*)

Definition at line 2125 of file database-oli.c.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    return DBlspVecGet(&(dataBase->lspVec), id);
}
```

4.11.3.15 long DBgetMaxNodeID (DataBase ∗ *dataBase*)

Definition at line 1454 of file database-oli.c.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nodeVec.top-1;
}
```

4.11.3.16 long DBgetNbLinks (DataBase ∗ *dataBase*)

Definition at line 1543 of file database-oli.c.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nbLinks;
}
```

4.11.3.17 long DBgetNbNodes (DataBase ∗ *dataBase*)

Definition at line 1442 of file database-oli.c.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nbNodes;
}
```

4.11.3.18 LongList∗ DBgetNodeInNeighb (DataBase ∗ *dataBase*, long *id*)

Definition at line 2209 of file database-oli.c.

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
    {
        addError(CRITICAL,"Node don’t exist (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return NULL;
    }

    return (&(node->inNeighb));
}
```

4.11.3.19 LongList∗ DBgetNodeOutNeighb (DataBase ∗ *dataBase*, long *id*)

Definition at line 2231 of file database-oli.c.

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
    {
        addError(CRITICAL,"Node don’t exist (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return NULL;
    }

    return (&(node->outNeighb));
}
```

4.11.3.20 int DBlinkStateCopy (DBLinkState ∗ *dst*, DBLinkState ∗ *src*)

Definition at line 660 of file database-oli.c.

```
{
    int i,j,ret=0;

    if (dst == NULL || src == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    dst->color=src->color;
    memcpy(&(dst->cap),&(src->cap),NB_OA * sizeof(double));
    memcpy(&(dst->rbw),&(src->rbw),NB_OA * NB_PREEMPTION * sizeof(double));
    memcpy(&(dst->pbw),&(src->pbw),NB_OA * NB_PREEMPTION * sizeof(double));

    for (i=0;(i<NB_OA && ret>=0);i++)
        for (j=0;(j<NB_PREEMPTION && ret>=0);j++)
        {
            ANDERROR(ret,dblVecCopy(&(dst->bbw[i][j]),&(src->bbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->remoteBbw[i][j]),&(src->remoteBbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->fbw[i][j]),&(src->fbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->remoteFbw[i][j]),&(src->remoteFbw[i][j])));
        }

    if (ret<0)
    {
        addError(CRITICAL,"Link state copy uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.11.3.21 int DBlinkStateDestroy (DBLinkState ∗ *ls*)

Definition at line 613 of file database-oli.c.

```
{
    int i,j;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            dblVecEnd(&(ls->bbw[i][j]));
            dblVecEnd(&(ls->remoteBbw[i][j]));
            dblVecEnd(&(ls->fbw[i][j]));
            dblVecEnd(&(ls->remoteFbw[i][j]));
        }
    free(ls);

    return 0;
}
```

4.11.3.22 int DBlinkStateEnd (DBLinkState ∗ *ls*)

Definition at line 637 of file database-oli.c.

```
{
    int i,j;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            dblVecEnd(&(ls->bbw[i][j]));
            dblVecEnd(&(ls->remoteBbw[i][j]));
            dblVecEnd(&(ls->fbw[i][j]));
            dblVecEnd(&(ls->remoteFbw[i][j]));
        }

    return 0;
}
```

4.11.3.23 int DBlinkStateInit (DBLinkState ∗ *ls*)

Definition at line 530 of file database-oli.c.

```
{
    int i,j,k,l;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(ls, 0, sizeof(DBLinkState));

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
            {
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                dblVecEnd(&(ls->fbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
        }

    return 0;
}
```

4.11.3.24 DBLinkState∗ DBlinkStateNew ()

Definition at line 444 of file database-oli.c.

Referenced by computeBackup().

```
{
    DBLinkState* ls;
    int i,j,k,l;

    if ((ls=calloc(1,sizeof(DBLinkState)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
            {
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                 __FILE__,__LINE__);
                return NULL;
            }
            else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return NULL;
            }
            else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return NULL;
            }
            else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                dblVecEnd(&(ls->fbw[i][j]));
                for (k=i;k>=0;k--)
```

```
                        for (l=j-1;l>=0;l--)
                        {
                            dblVecEnd(&(ls->bbw[k][l]));
                            dblVecEnd(&(ls->remoteBbw[k][l]));
                            dblVecEnd(&(ls->fbw[k][l]));
                            dblVecEnd(&(ls->remoteFbw[k][l]));
                        }
                    free(ls);
                    addError(CRITICAL,"Unable to create link state in %s at line %d",
                             __FILE__,__LINE__);
                    return NULL;
                }
            }

    return ls;
}
```

4.11.3.25 int DBlspCompare (const DBLabelSwitchedPath ∗ *LSPa*, const DBLabelSwitchedPath ∗ *LSPb*)

Definition at line 357 of file database-oli.c.

```
{
    if (LSPa->precedence > LSPb->precedence)
        return 1;
    else if (LSPa->precedence < LSPb->precedence)
        return -1;
    else if (LSPa->bw[0] > LSPb->bw[0])
        return 1;
    else if (LSPa->bw[0] < LSPb->bw[0])
        return -1;
    else
    {
        if (LSPa->id < LSPb->id)
            return 1;
        else if (LSPa->id > LSPb->id)
            return -1;
    }

    return 0;
}
```

4.11.3.26 int DBlspCopy (DBLabelSwitchedPath ∗ *dst*, DBLabelSwitchedPath ∗ *src*)

Definition at line 157 of file database-oli.c.

```
{
    int ret=0;

    if (dst == NULL || src==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    dst->id=src->id;
    dst->precedence=src->precedence;
    memcpy(dst->bw,src->bw, NB_OA * sizeof(double));
    dst->noContentionId = src->noContentionId;
    ANDERROR(ret,longListCopy(&(dst->forbidLinks),&(src->forbidLinks)));
    ANDERROR(ret,longListCopy(&(dst->path),&(src->path)));
    dst->type=src->type;
    dst->primID=src->primID;
    ANDERROR(ret,longListCopy(&(dst->primPath),&(src->primPath)));
    ANDERROR(ret,longListCopy(&(dst->backLSPIDs),&(src->backLSPIDs)));

    if (ret<0)
    {
        addError(CRITICAL,"Label switched path copy uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.11.3.27 int DBlspDestroy (DBLabelSwitchedPath * *lsp*)

Definition at line 122 of file database-oli.c.

```
{
    if (lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListEnd(&(lsp->backLSPIDs));
    longListEnd(&(lsp->primPath));
    longListEnd(&(lsp->path));
    longListEnd(&(lsp->forbidLinks));
    free(lsp);

    return 0;
}
```

4.11.3.28 int DBlspEnd (DBLabelSwitchedPath * *lsp*)

Definition at line 140 of file database-oli.c.

```
{
    if (lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListEnd(&(lsp->backLSPIDs));
    longListEnd(&(lsp->primPath));
    longListEnd(&(lsp->path));
    longListEnd(&(lsp->forbidLinks));

    return 0;
}
```

4.11.3.29 int DBlspInit (DBLabelSwitchedPath ∗ *lsp*)

Definition at line 73 of file database-oli.c.

```
{
    if (lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->forbidLinks),-1)<0)
    {
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->path),-1)<0)
    {
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->primPath),-1)<0)
    {
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->backLSPIDs),-1)<0)
    {
        longListEnd(&(lsp->primPath));
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(lsp->bw, 0, NB_OA * sizeof(double));
    lsp->noContentionId=-1;   //very important

    return 0;
}
```

4.11.3.30 int DBlspListDestroy (DBLSPList ∗ *list*)

Definition at line 251 of file database-oli.c.

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(list->cont);
    free(list);

    return 0;
}
```

4.11.3.31 int DBlspListEnd (DBLSPList ∗ *list*)

Definition at line 266 of file database-oli.c.

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(list->cont);
    list->cont = NULL;
    list->size = 0;
    list->top = 0;

    return 0;
}
```

4.11.3.32 int DBlspListInit (DBLSPList ∗ *list*, long *size*)

Definition at line 223 of file database-oli.c.

```
{
    void* ptr=NULL;

    if (list == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LSPLIST_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    list->size = size;
    list->top = 0;
    list->cont = ptr;

    return 0;
}
```

4.11.3.33 int DBlspListInsert (DBLSPList ∗ *list*, DBLabelSwitchedPath ∗ *lsp*)

Definition at line 283 of file database-oli.c.

```
{
    int a,b;
    void *ptr=NULL;

    if (list == NULL || list->cont == NULL || lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // check the capacity of the list
    if (list->top >= list->size)
    {
        if ((ptr = realloc(list->cont, list->size
                           * 2 * sizeof(DBLabelSwitchedPath*))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            list->cont=ptr;
            list->size*=2;
        }
    }

    // find the position in the list (to keep it sorted)
    a = 0;
    b = list->top-1;

    // empty list or after the last elem
    if (list->top == 0 || DBlspCompare(list->cont[b], lsp) >= 0)
    {
        list->cont[list->top++] = lsp;
        return (list->top-1);
    }

    // before the first elem
    if (DBlspCompare(lsp, list->cont[a]) >= 0)
    {
        memmove(list->cont+1, list->cont, (list->top)*sizeof(void*));
        list->cont[0] = lsp;
        list->top++;
        return 0;
    }

    // now the insert position is inside ]a,b[
    while (b - a > 1)
    {
        int mid = (a + b)/2;
        int ret = DBlspCompare(lsp, list->cont[mid]);

        if (ret == 1)
            b = mid;
        else if (ret == -1)
            a = mid;
        else // if (ret == 0)
        {
            a = mid;
            b = mid;
        }
    }

    // now insert before b
    memmove(list->cont+b+1, list->cont+b, (list->top - b)*sizeof(void*));
    list->cont[b] = lsp;
    list->top++;

    return b;
}
```

4.11.3.34 DBLSPList∗ DBlspListNew (long *size*)

Definition at line 193 of file database-oli.c.

```
{
    DBLSPList *list=NULL;
    void* ptr=NULL;

    if ((list = calloc(1,sizeof(DBLSPList))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LSPLIST_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(list);
        return NULL;
    }

    list->size = size;
    list->top = 0;
    list->cont = ptr;

    return list;
}
```

4.11.3.35 int DBlspListRemove (DBLSPList ∗ *list*, DBLabelSwitchedPath ∗ *lsp*)

Definition at line 378 of file database-oli.c.

```
{
    int a,b,index;

    if (list == NULL || list->cont == NULL || lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // find the position in the list
    a = 0;
    b = list->top-1;

    // empty list
    if (list->top == 0)
    {
        addError(WARNING,"Removing inexistent LSP in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    while (b - a > 1)
    {
        int mid = (a + b)/2;
        int ret = DBlspCompare(lsp, list->cont[mid]);

        if (ret == 1)
            b = mid;
        else if (ret == -1)
            a = mid;
        else // if (ret == 0)
        {
            a = mid;
            b = mid;
        }
    }

    if (DBlspCompare(lsp, list->cont[a]) == 0)
    {
        index = a;
    }
    else if (DBlspCompare(lsp, list->cont[b]) == 0)
    {
        index = b;
    }
    else // not found
    {
        addError(WARNING,"Removing inexistent LSP in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // now delete index
    memmove(list->cont + index, list->cont + index + 1, (list->top - index -1)*sizeof(void*));
    list->top--;

    return 0;
}
```

4.11.3.36 DBLabelSwitchedPath* DBlspNew ()

Definition at line 19 of file database-oli.c.

Referenced by DBaddLSP(), and evalLS().

```
{
    DBLabelSwitchedPath* lsp;

    if ((lsp=calloc(1,sizeof(DBLabelSwitchedPath)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->forbidLinks),-1)<0)
    {
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->path),-1)<0)
    {
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->primPath),-1)<0)
    {
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->backLSPIDs),-1)<0)
    {
        longListEnd(&(lsp->primPath));
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    lsp->noContentionId=-1; //very important

    return lsp;
}
```

4.11.3.37 DataBase∗ DBnew (long *ID*)

Definition at line 1280 of file database-oli.c.

```
{
    DataBase *dataBase=NULL;

    if ((dataBase=calloc(1,sizeof(DataBase)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    dataBase->id=ID;

    if (DBnodeVecInit(&(dataBase->nodeVec),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general node container in %s at line %d",
                 __FILE__,__LINE__);
        free(dataBase);
        return NULL;
    }

    if (DBlspVecInit(&(dataBase->lspVec),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general LSP container in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        free(dataBase);
        return NULL;
    }

    if (DBlinkTabInit(&(dataBase->linkTab),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general link container in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        free(dataBase);
        return NULL;
    }

    if (longVecInit(&(dataBase->linkSrcVec),LINKTAB_INITSIZE)<0)
    {
        addError(CRITICAL,"Unable to initialize the link id-src translater in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        DBlinkTabEnd(&(dataBase->linkTab));
        free(dataBase);
        return NULL;
    }

    if (longVecInit(&(dataBase->linkDstVec),LINKTAB_INITSIZE)<0)
    {
        addError(CRITICAL,"Unable to initialize the link id-dst translater in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        DBlinkTabEnd(&(dataBase->linkTab));
        longVecEnd(&(dataBase->linkSrcVec));
        free(dataBase);
        return NULL;
    }

    dataBase->nbNodes=0;
    dataBase->nbLinks=0;

    return dataBase;
}
```

4.11.3.38 void DBprintDB (DataBase ∗ *db*)

Definition at line 2253 of file database-oli.c.

```
{
    long i,j;

    printf("Printing info about nodes ...\n");
    printf("----------------------------\n");

    for (i=0; i<db->nodeVec.size; i++)
    {
        if (db->nodeVec.cont[i])
        {
            printf("Node id : %ld\n", i);
            printf("------------\n");
            DBprintNode(db->nodeVec.cont[i]);
        }
    }

    printf("\nPrinting info about links ...\n");
    printf("----------------------------\n");

    for (i=0; i<db->linkTab.size; i++)
        for (j=0; j<db->linkTab.size; j++)
        {
            if (db->linkTab.cont[i][j])
            {
                printf("Link %ld-%ld (id = %ld)\n", i, j, DBgetLinkID(db, i, j));
                printf("----------------------\n");

                DBprintLink(db->linkTab.cont[i][j]);

            }
        }
}
```

4.11.3.39 int DBremoveLink (DataBase * *dataBase*, long *src*, long *dst*)

Definition at line 1635 of file database-oli.c.

```
{
    int id,ret=0;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((DBnodeVecGet(&(dataBase->nodeVec),src)==NULL) ||
        (DBnodeVecGet(&(dataBase->nodeVec),dst)==NULL) ||
        (DBlinkTabGet(&(dataBase->linkTab),src,dst)==NULL))
    {
        addError(CRITICAL,"Link doesn't exist or database unconsistancy in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
    ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[dst]->inNeighb),src));

    ANDERROR(ret,DBlinkTabRemove(&(dataBase->linkTab),src,dst));

    id=DBgetLinkID(dataBase,src,dst);
    ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,0));
    ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,0));

    while (dataBase->linkSrcVec.cont[dataBase->linkSrcVec.top-1] == 0)
        dataBase->linkSrcVec.top--;

    if (ret<0)
    {
        addError(CRITICAL,"Link removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks--;

    return ret;
}
```

4.11.3.40 int DBremoveLSP (DataBase ∗ *dataBase*, long *id*)

Definition at line 2005 of file database-oli.c.

```
{
    DBLabelSwitchedPath *lsp=NULL, *contentLSP=NULL;
    int i,ret=0;
    DBLink *lnk=NULL;
    LongVec isProcessed;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lsp = DBlspVecGet(&(dataBase->lspVec), id)) == NULL)
    {
        addError(CRITICAL,"Trying to remove inexistent LSP (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return -1;
    }

    if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
    {
        addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

#if defined SIMULATOR
    // Remove the LSP from each link list and update all the linkstates
    for (i=0;i<lsp->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
        ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->path.cont[i],
                                 lsp->path.cont[i+1], &(lnk->state), lsp));
        isProcessed.cont[lnk->id] = 1;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;i<lsp->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->primPath.cont[i],
                               lsp->primPath.cont[i+1], &(lnk->state), lsp));
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
#elif defined AGENT
    // Remove the LSP to the link attached to the agent and update the linkstate
    for (i=0;i<lsp->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));

        if (lsp->path.cont[i] == dataBase->id)
        {
            ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->path.cont[i],
                                           lsp->path.cont[i+1], &(lnk->state), lsp));
            isProcessed.cont[lnk->id] = 1;
        }
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;i<lsp->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);

            if (lsp->primPath.cont[i] == dataBase->id)
            {
                if (isProcessed.cont[lnk->id] == 0)
                {
                    ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->primPath.cont[i],
                                                   lsp->primPath.cont[i+1], &(lnk->state), lsp));
                }
                break;
            }
        }
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    longVecEnd(&(isProcessed));

    // remove the lsp from the global list
    ANDERROR(ret,DBlspVecRemove(&(dataBase->lspVec), id));

    if (lsp->noContentionId>=0)
    {
        if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),lsp->noContentionId))==NULL)
        {
            addError(WARNING,"Unable to get no contention LSP in %s at line %d",
                     __FILE__,__LINE__);
            // not critical enough to abort
        }
        contentLSP->noContentionId=-1;
    }

    // free the lsp
    DBlspDestroy(lsp);

    if (ret<0)
    {
        addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.11.3.41 int DBremoveNode (DataBase ∗ *dataBase*, long *id*)

Definition at line 1499 of file database-oli.c.

```
{
    DBNode *node=NULL;
    int ret=0;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec),id)) == NULL)
    {
        addError(CRITICAL,"Trying to remove an inexistent node in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // remember that DBremoveLink will update the neighbour list
    while(node->inNeighb.top > 0)
    {
        ANDERROR(ret,DBremoveLink(dataBase,node->inNeighb.cont[node->inNeighb.top-1],id));
    }

    // remember that DBremoveLink will update the neighbour list
    while(node->outNeighb.top > 0)
    {
        ANDERROR(ret,DBremoveLink(dataBase,id,node->outNeighb.cont[node->outNeighb.top-1]));
    }

    ANDERROR(ret,DBnodeVecRemove(&(dataBase->nodeVec),id));

    if (ret<0)
    {
        addError(CRITICAL,"Node removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks--;

    return ret;
}
```

4.11.3.42 int DBsetLinkState (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *newLS*)

Definition at line 2180 of file database-oli.c.

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL || newLS == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return -1;
    }

2198     if (DBlinkStateCopy(&(lnk->state), newLS)<0)
2199     {
2200         addError(CRITICAL,"Impossible to set linkstate on link (src = %ld, dst = %ld) in %s at line %
2201                   src,dst,__FILE__,__LINE__);
2202         return -1;
2203     }
2204 
2205     return 0;
2206 }
```

4.11.3.43 int DBupdateLSOnRemove (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *ls*, DBLabelSwitchedPath ∗ *lsp*)

Definition at line 1269 of file database-oli.c.

```
1270 {
1271     return updateLS(dataBase, src, dst, ls, lsp, REMOVE);
1272 }
```

4.11.3.44 int DBupdateLSOnSetup (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *ls*, DBLabelSwitchedPath ∗ *lsp*)

Definition at line 1264 of file database-oli.c.

```
1265 {
1266     return updateLS(dataBase, src, dst, ls, lsp, SETUP);
1267 }
```

4.11.3.45 int evalLS (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *newLS*, DBLinkState ∗ *oldLS*, LSPRequest ∗ *req*, operation *op*)

Definition at line 838 of file database-oli.c.

References addError(), LSPRequest_::bw, DBLabelSwitchedPath_::bw, CRITICAL, DBgetLSP(), DBlinkStateCopy(), DBlspDestroy(), DBlspNew(), LSPRequest_::forbidLinks, DBLabelSwitchedPath_::forbidLinks, GLOBAL_BACK, LSPrerouteInfo_::id, LOCAL_BACK, longListCopy, NB_OA, DBLabelSwitchedPath_::noContentionId, LSPRequest_::path, DBLabelSwitchedPath_::path, LSPRequest_::precedence, DBLabelSwitchedPath_::precedence, PRIM, LSPRequest_::primID, DBLabelSwitchedPath_::primID, DBLabelSwitchedPath_::primPath, LSPRequest_::rerouteInfo, LongVec_::top, DBLabelSwitchedPath_::type, LSPRequest_::type, and updateLS().

```
839 {
840     DBLabelSwitchedPath* lsp, *primLSP;
841     int ret;
842 
843     // check the arguments
844     if ((dataBase==NULL) || (newLS==NULL) || (oldLS==NULL) || (req==NULL))
845     {
846         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
847                   __FILE__,__LINE__);
848         return -1;
849     }
850 
851     // duplicate the LS
852     if (newLS != oldLS && DBlinkStateCopy(newLS, oldLS) < 0)
```

```
    {
        addError(CRITICAL,"Impossible to duplicate the linkState in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // now build a false LSP satisfying the request ....
    lsp = DBlspNew();

    lsp->precedence = req->precedence;
    memcpy(lsp->bw, req->bw, NB_OA * sizeof(double));
    longListCopy(&(lsp->forbidLinks), &(req->forbidLinks));

    if (req->rerouteInfo.id >= 0)
    {
        lsp->noContentionId = req->rerouteInfo.id;
    }

    switch(req->type)
    {
        case PRIM:
            lsp->type = PRIM;
            lsp->primID = -1;
            break;

        case GLOBAL_BACK:
        case LOCAL_BACK:
            lsp->type = req->type;
            lsp->primID = req->primID;

            // look up the primary path ....
            if ((primLSP = DBgetLSP(dataBase, lsp->primID)) == NULL)
            {
                addError(CRITICAL,"Impossible to determine the primary path in %s at line %d",
                 __FILE__,__LINE__);
                DBlspDestroy(lsp);
                return -1;
            }

            longListCopy(&(lsp->primPath), &(primLSP->path));

            break;

        default:
            addError(CRITICAL,"Unknown request type (NULL) in %s at line %d",
                 __FILE__,__LINE__);
            DBlspDestroy(lsp);
            return -1;
    }

    if (req->path.top < 2)
    {
        addError(CRITICAL,"Wrong path in request in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(lsp);
        return -1;
    }

    if (longListCopy(&(lsp->path), &(req->path)) < 0)
    {
        addError(CRITICAL,"Impossible to duplicate path in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(lsp);
        return -1;
    }

    ret = updateLS(dataBase, src, dst, newLS, lsp, op);

    // clean up ....
    DBlspDestroy(lsp);

    return ret;
}
```

4.11.3.46 int updateLS (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗, operation)

Definition at line 927 of file database-oli.c.

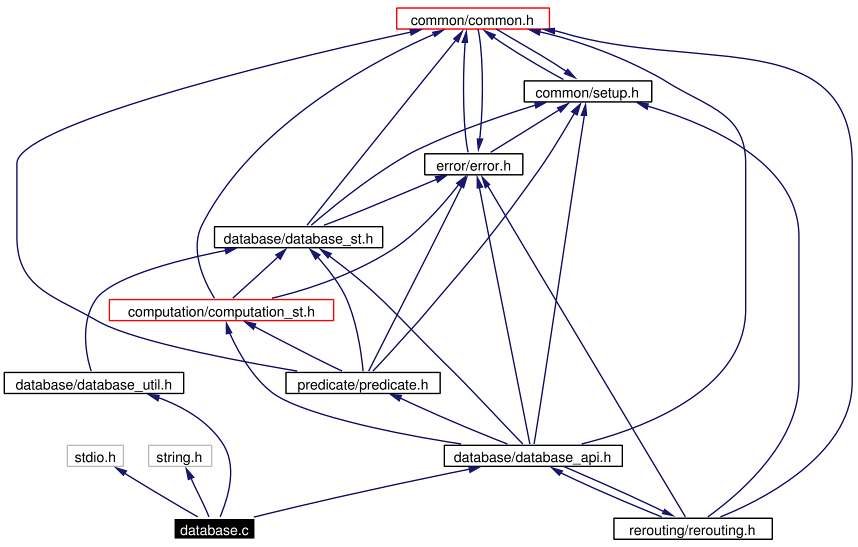
References addError(), DBLinkState_::bbw, DBLabelSwitchedPath_::bw, computeRBW(), LongVec_::cont, CRITICAL, DBgetLinkID(), DBgetLSP(), dblVecResize(), FALSE, DBLinkState_::fbw, GLOBAL_BACK, LOCAL_BACK, max, NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, DBLinkState_::pbw, DBLabelSwitchedPath_::precedence, DBLabelSwitchedPath_::primPath, DBLinkState_::rbw, REMOVE, SETUP, LongVec_::top, TRUE, DBLabelSwitchedPath_::type, and WARNING.

```
{
    bool path = FALSE;
    int myPosPath=-1, myPosPrimPath=-1;
    bool primPath = FALSE;
    int i,oa;
    int mult;
    int plink;
    double newBW[NB_OA];
    DBLabelSwitchedPath* oldLSP=NULL;
    bool rerouting = FALSE;

    // check the arguments
    if ((dataBase==NULL) || (ls==NULL) || (lsp==NULL))
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // am i on the path ?
    for (i=0; i<lsp->path.top; ++i)
    {
        if (lsp->path.cont[i] == src)
            break;
    }

    if (i < (lsp->path.top - 1) && lsp->path.cont[i+1] == dst)
    {
        path = TRUE;
        myPosPath = i;
    }

    // if i am on the path, am i the a rerouted LSP sharing the link with a preempted one ?
    if (path == TRUE && lsp->noContentionId >= 0)
    {
        if ((oldLSP = DBgetLSP(dataBase, lsp->noContentionId)) == NULL)
        {
            addError(WARNING,"Cannot find the old LSP in %s at line %d",
                 __FILE__,__LINE__);
        }
        else
        {
            for (i=0; i<oldLSP->path.top; ++i)
            {
                if (oldLSP->path.cont[i] == src)
                    break;
            }

            if (i < (oldLSP->path.top - 1) && oldLSP->path.cont[i+1] == dst)
            {
                rerouting = TRUE;
            }
        }
    }

    // if rerouting -> check if there is a change in the bandwidth reservation. if no change we can le
    if (rerouting == TRUE)
    {
        bool test=FALSE;


        for (i=0; i<NB_OA; i++) {
            if ((newBW[i] = max(lsp->bw[i] - oldLSP->bw[i], 0)) != 0)
                test = TRUE;
        }

        if (test == FALSE)
        {
            return 0;
        }
    }

    // if it is a backup am i on the prim path ?
    if (lsp->type == LOCAL_BACK || lsp->type == GLOBAL_BACK)
    {
        for (i=0; i<lsp->primPath.top; ++i)
        {
            if (lsp->primPath.cont[i] == src)
                break;
        }

        if (i < (lsp->primPath.top - 1) && lsp->primPath.cont[i+1] == dst)
        {
            primPath = TRUE;
            myPosPrimPath = i;
        }
    }

    if (!path && !primPath) // not concerned by this update ...
        return 0;

    if (op == SETUP)
    {
        mult = 1;
    }
    else if (op == REMOVE)
    {
        mult = -1;
    }
    else
    {
        addError(CRITICAL,"Bad argument (unknown operation) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (lsp->type == LOCAL_BACK)
    {
        // the path is a local backup
        // -------------------------
        int start,end;

        // which link are we protecting ... and the start
        for (i=0; i<lsp->primPath.top; ++i)
        {
            if (lsp->primPath.cont[i] == lsp->path.cont[0])
                break;
        }

        if (i < lsp->primPath.top - 1)
        {
            plink = DBgetLinkID(dataBase, lsp->path.cont[0], lsp->primPath.cont[i+1]);
            start = i;
        }
        else
        {
            addError(CRITICAL,"Cannot determine the link to protect in %s at line %d",
                 __FILE__,__LINE__);
            return -1;
        }

        if (path == TRUE)
        {
            // update bbw
            for (oa=0; oa<NB_OA; ++oa)
            {
                if (ls->bbw[oa][lsp->precedence].size <= plink)
                    dblVecResize(&(ls->bbw[oa][lsp->precedence]), plink+1);

                if (rerouting == FALSE)
                    ls->bbw[oa][lsp->precedence].cont[plink] += (mult * lsp->bw[oa]);
                else
                    ls->bbw[oa][lsp->precedence].cont[plink] += (mult * newBW[oa]);
            }
        }

        if (primPath == TRUE)
        {
            // find the end
            for (i=start; i<lsp->primPath.top; ++i)
            {
                if (lsp->primPath.cont[i] == lsp->path.cont[lsp->path.top - 1])
                    break;
            }

            if (i < lsp->primPath.top)
            {
                end = i;
            }
            else
            {
                // finding an end is not required during path evaluation
                end = -1;
                /*
                addError(CRITICAL,"Cannot determine the merging point in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
                */
            }

            // maybe fbw must be recomputed
            // if i’m before path.cont[0] or after path.cont[last]
            if (myPosPrimPath < start || myPosPrimPath >= end)
            {
                for (oa=0; oa<NB_OA; ++oa)
                {
                    if (ls->fbw[oa][lsp->precedence].size <= plink)
                        dblVecResize(&(ls->fbw[oa][lsp->precedence]), plink+1);

                    if (rerouting == FALSE)
                        ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * lsp->bw[oa]);
                    else
                        ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * newBW[oa]);
                }
            }
        }
    }
    else if (lsp->type == GLOBAL_BACK)
    {
        // the path is a end-to-end backup
        // ------------------------------
        int start,end;

        // we are protecting all nodes between path.cont[0] and path.cont[end]
        // find the start
        for (i=0; i<lsp->primPath.top; ++i)
        {
            if (lsp->primPath.cont[i] == lsp->path.cont[0])
                break;
        }

        if (i < lsp->primPath.top)
        {
            start = i;
        }
        else
        {
            addError(CRITICAL,"Cannot determine the link to protect in %s at line %d",
                 __FILE__,__LINE__);
            return -1;
        }

        // find the end
        end = lsp->primPath.top - 1;

        if (path == TRUE)
        {
            // update bbw
            for (i=start; i<end; ++i)
            {
                if ((plink = DBgetLinkID(dataBase, lsp->primPath.cont[i], lsp->primPath.cont[i+1])) >
                    for (oa=0; oa<NB_OA; ++oa)
                    {
                        if (ls->bbw[oa][lsp->precedence].size <= plink)
                            dblVecResize(&(ls->bbw[oa][lsp->precedence]), plink+1);

                        if (rerouting == FALSE)
                            ls->bbw[oa][lsp->precedence].cont[plink] += (mult * lsp->bw[oa]);
                        else
                            ls->bbw[oa][lsp->precedence].cont[plink] += (mult * newBW[oa]);
                    }
                else
                {
                    addError(CRITICAL,"Cannot determine link ID in %s at line %d",
                             __FILE__,__LINE__);
                    return -1;
                }
            }
        }

        if (primPath == TRUE)
        {
            // maybe fbw must be recomputed
            // if i’m before path.cont[0] or after path.cont[last]

            for (i=0; i<start; ++i)
            {
                if ((plink = DBgetLinkID(dataBase, lsp->primPath.cont[i], lsp->primPath.cont[i+1])) >
                    for (oa=0; oa<NB_OA; ++oa)
                    {
                        if (ls->fbw[oa][lsp->precedence].size <= plink)
                            dblVecResize(&(ls->fbw[oa][lsp->precedence]), plink+1);

                        if (rerouting == FALSE)
                            ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * lsp->bw[oa]);
                        else
                            ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * newBW[oa]);
                    }
                else
                {
                    addError(CRITICAL,"Cannot determine link ID in %s at line %d",
                             __FILE__,__LINE__);
                    return -1;
                }
            }

            for (i=end; i<lsp->primPath.top-1; ++i)
            {
                plink = DBgetLinkID(dataBase, lsp->primPath.cont[i], lsp->primPath.cont[i+1]);
                for (oa=0; oa<NB_OA; ++oa)
                {
                    if (ls->fbw[oa][lsp->precedence].size <= plink)
                        dblVecResize(&(ls->fbw[oa][lsp->precedence]), plink+1);

                    if (rerouting == FALSE)
                        ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * lsp->bw[oa]);
                    else
                        ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * newBW[oa]);
                }
            }
        }
    }
    else
    {
        // path is a primary

        // update pbw
        for (i=0; i<NB_OA; ++i)
        {
            if (rerouting == FALSE)
                ls->pbw[i][lsp->precedence] += (mult * lsp->bw[i]);
            else
                ls->pbw[i][lsp->precedence] += (mult * newBW[i]);
        }

        // update fbw
        for (i=0; i<lsp->path.top - 1; ++i)
        {
            int id;

            if ((id = DBgetLinkID(dataBase, lsp->path.cont[i], lsp->path.cont[i+1])) >= 0)
                for (oa=0; oa<NB_OA; ++oa)
                {
                    if (ls->fbw[oa][lsp->precedence].size <= id)
                        dblVecResize(&(ls->fbw[oa][lsp->precedence]), id+1);

                    if (rerouting == FALSE)
                        ls->fbw[oa][lsp->precedence].cont[id] += (mult * lsp->bw[oa]);
                    else
                        ls->fbw[oa][lsp->precedence].cont[id] += (mult * newBW[oa]);
                }
            else
            {
                addError(CRITICAL,"Cannot determine link ID in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
        }
    }

    computeRBW(dataBase, ls->rbw, ls->pbw, ls->bbw, ls->fbw);

    return 0;
}
```

4.12 database.c File Reference

```
#include "database/database_api.h"
#include "database/database_util.h"
#include <stdio.h>
#include <string.h>
```

Include dependency graph for database.c:



Typedefs

- typedef enum operation_ operation

Enumerations

- enum operation_ { SETUP, REMOVE }

Functions

- DBLabelSwitchedPath ∗ DBlspNew ()
- int DBlspInit (DBLabelSwitchedPath ∗lsp)
- int DBlspDestroy (DBLabelSwitchedPath ∗lsp)
- int DBlspEnd (DBLabelSwitchedPath ∗lsp)
- int DBlspCopy (DBLabelSwitchedPath ∗dst, DBLabelSwitchedPath ∗src)
- DBLSPList ∗ DBlspListNew (long size)
- int DBlspListInit (DBLSPList ∗list, long size)
- int DBlspListDestroy (DBLSPList ∗list)
- int DBlspListEnd (DBLSPList ∗list)
- int DBlspListInsert (DBLSPList ∗list, DBLabelSwitchedPath ∗lsp)
- int DBlspCompare (const DBLabelSwitchedPath ∗LSPa, const DBLabelSwitchedPath ∗LSPb)

- int DBlspListRemove (DBLSPList ∗list, DBLabelSwitchedPath ∗lsp)
- DBLinkState ∗ DBlinkStateNew ()
- int DBlinkStateInit (DBLinkState ∗ls)
- int DBlinkStateDestroy (DBLinkState ∗ls)
- int DBlinkStateEnd (DBLinkState ∗ls)
- int DBlinkStateCopy (DBLinkState ∗dst, DBLinkState ∗src)
- int computeRBW (DataBase ∗dataBase, double rbw[NB_OA][NB_PREEMPTION], double pbw[NB_OA][NB_PREEMPTION], DoubleVec bbw[NB_OA][NB_PREEMPTION], DoubleVec fbw[NB_OA][NB_PREEMPTION])
- int updateLS (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗, operation)
- int evalLS (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS, DBLinkState ∗oldLS, LSPRequest ∗req, operation op)
- int DBevalLSOnSetup (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS, DBLinkState ∗oldLS, LSPRequest ∗req)
- int DBevalLSOnRemove (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS, DBLinkState ∗oldLS, LSPRequest ∗req)
- int DBupdateLSOnSetup (DataBase ∗dataBase, long src, long dst, DBLinkState ∗ls, DBLabelSwitchedPath ∗lsp)
- int DBupdateLSOnRemove (DataBase ∗dataBase, long src, long dst, DBLinkState ∗ls, DBLabelSwitchedPath ∗lsp)
- DataBase ∗ DBnew (long ID)
- int DBdestroy (DataBase ∗dataBase)
- long DBgetID (DataBase ∗dataBase)
- long DBgetLinkID (DataBase ∗dataBase, long src, long dst)
- long DBgetLinkSrc (DataBase ∗dataBase, long id)
- long DBgetLinkDst (DataBase ∗dataBase, long id)
- long DBgetNbNodes (DataBase ∗dataBase)
- long DBgetMaxNodeID (DataBase ∗dataBase)
- int DBaddNode (DataBase ∗dataBase, long id)
- int DBremoveNode (DataBase ∗dataBase, long id)
- long DBgetNbLinks (DataBase ∗dataBase)
- int DBaddLink (DataBase ∗dataBase, long id, long src, long dst, DBLinkState ∗initLinkState)
- int DBremoveLink (DataBase ∗dataBase, long src, long dst)
- int DBaddLSP (DataBase ∗dataBase, DBLabelSwitchedPath ∗lsp, LongList ∗preemptList)
- int DBremoveLSP (DataBase ∗dataBase, long id)
- DBLabelSwitchedPath ∗ DBgetLSP (DataBase ∗dataBase, long id)
- DBLSPList ∗ DBgetLinkLSPs (DataBase ∗dataBase, long src, long dst)
- DBLinkState ∗ DBgetLinkState (DataBase ∗dataBase, long src, long dst)
- int DBsetLinkState (DataBase ∗dataBase, long src, long dst, DBLinkState ∗newLS)
- LongList ∗ DBgetNodeInNeighb (DataBase ∗dataBase, long id)
- LongList ∗ DBgetNodeOutNeighb (DataBase ∗dataBase, long id)
- void DBprintDB (DataBase ∗db)

4.12.1 Typedef Documentation

4.12.1.1 typedef enum operation_ operation

4.12.2 Enumeration Type Documentation

4.12.2.1 enum operation_

Enumeration values:

SETUP

REMOVE

Definition at line 834 of file database.c.

```
834 { SETUP, REMOVE} operation;
```

4.12.3 Function Documentation

4.12.3.1 int computeRBW (DataBase ∗ *dataBase*, double *rbw*[NB_OA][NB_PREEMPTION], double *pbw*[NB_OA][NB_PREEMPTION], DoubleVec *bbw*[NB_OA][NB_PREEMPTION], DoubleVec *fbw*[NB_OA][NB_PREEMPTION])

Definition at line 694 of file database.c.

References addError(), CRITICAL, NB_OA, and NB_PREEMPTION.

Referenced by updateLS().

```
696 {
697 #if defined LINUX && defined TIME1
698     struct timezone tz;
699     struct timeval  t1,t2;
700 #endif
701     int nbLink = 0, seenLinks;
702     int nbNode = 0, seenNodes;
703     int i,oa,p;
704     DoubleVec* gbw;
705     double totBbw = 0;
706     double totFbw = 0;
707     double m, oldM;
708
709     if ((rbw==NULL) || (pbw==NULL) || (bbw==NULL) || (fbw==NULL))
710     {
711         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
712                  __FILE__,__LINE__);
713         return -1;
714     }
715
716 #if defined LINUX && defined TIME1
717     gettimeofday(&t1, &tz);
718 #endif
719
720 /*
721     nbLink = dataBase->linkSrcVec.top;
722     nbNode = dataBase->nodeVec.top;
723
724     if ((gbw = calloc(nbLink + nbNode, sizeof(DoubleVec))) == NULL)
725     {
726         addError(CRITICAL,"Cannot allocate GBW in %s at line %d",
727                  __FILE__,__LINE__);
728         return -1;
729     }
730
731     for (i=0; i<nbLink + nbNode; ++i)
732         dblVecInit(&(gbw[i]), NB_PREEMPTION);
733
734     for (oa=0; oa<NB_OA; ++oa)
735     {
736         seenLinks = 0;
737
738         // phase 1a (links)
739         for (i=0; seenLinks<dataBase->nbLinks; ++i)
740         {
```

```
            if (dataBase->linkSrcVec.cont[i] == 0)
                continue;
            else
                seenLinks++;

            totBbw = 0;
            totFbw = 0;

            for(p=0; p<NB_PREEMPTION; ++p)
            {
                if (i < bbw[oa][p].size)
                    totBbw += bbw[oa][p].cont[i];
                if (i < fbw[oa][p].size)
                    totFbw += fbw[oa][p].cont[i];

                gbw[i].cont[p] = max(0, totBbw - totFbw);
            }
        }

        seenNodes = 0;

        // phase 1b (nodes or any set of links)
        for (i=0; seenNodes<dataBase->nbNodes; ++i)
        {
            if (dataBase->nodeVec.cont[i] == NULL)
                continue;
            else
                seenNodes++;

            totBbw = 0;
            totFbw = 0;

            for(p=0; p<NB_PREEMPTION; ++p)
            {
                LongList* lst;
                if ((lst = DBgetNodeInNeighb(dataBase, i)) != NULL)
                {
                    int l;
                    for (l=0; l<lst->top; ++l)
                    {
                        int lnkID = DBgetLinkID(dataBase, lst->cont[l], i);
                        if (lnkID < bbw[oa][p].size)
                            totBbw += bbw[oa][p].cont[lnkID];
                        if (lnkID < fbw[oa][p].size)
                            totFbw += fbw[oa][p].cont[lnkID];
                    }
                }

                gbw[i + nbLink].cont[p] = max(0, totBbw - totFbw);
            }
        }

        // phase 2
        oldM = 0;

        for (p=0; p<NB_PREEMPTION; ++p)
        {
            m = 0;
            for (i=0; i<nbLink + nbNode; ++i)
            {
                if (gbw[i].cont[p] > m)
                {
                    m = gbw[i].cont[p];
                }
            }

            rbw[oa][p] = pbw[oa][p] + m - oldM;
            oldM = m;
        }
    }

    for (i=0; i<nbLink + nbNode; ++i)
        dblVecEnd(&(gbw[i]));

    free(gbw);
*/

    for (oa=0; oa<NB_OA; ++oa)
        for (p=0; p<NB_PREEMPTION; ++p)
        {
            rbw[oa][p] = pbw[oa][p];
        }

#if defined LINUX && defined TIME1
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time to compute rbw : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif

    return 0;

}
```

4.12.3.2 int DBaddLink (DataBase ∗ *dataBase*, long *id*, long *src*, long *dst*, DBLinkState ∗ *initLinkState*)

Definition at line 1612 of file database.c.

References addError(), ANDERROR, DBNodeVec_::cont, LongVec_::cont, CRITICAL, DBlinkDestroy(), DBlinkNew(), DBlinkStateCopy(), DBlinkTabSet(), DBnodeVecGet, DBLink_::id, DBNode_::inNeighb, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, longListPushBack, longListSort(), longVecSet(), max, DataBase_::nbLinks, DataBase_::nodeVec, DBNode_::outNeighb, LongVec_::size, DBLink_::state, and LongVec_::top.

```
{
    DBLink* link=NULL;
    int ret=0;

    if (dataBase == NULL || initLinkState==NULL
        || id <0 || src<0 || dst<0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (((id<dataBase->linkSrcVec.size) && (dataBase->linkSrcVec.cont[id]>0))
        ||
        ((id<dataBase->linkDstVec.size) && (dataBase->linkDstVec.cont[id]>0)))
    {
        addError(CRITICAL,"Trying to add a link with a reserved ID (ID=%ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return -1;
    }

    if ((link = DBlinkNew()) == NULL)
    {
        addError(CRITICAL,"Unable to create link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    link->id=id;

    if (DBlinkStateCopy(&(link->state), initLinkState))
    {
        addError(CRITICAL,"Unable to create link in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    if ((DBnodeVecGet(&(dataBase->nodeVec),src) == NULL) ||
        (DBnodeVecGet(&(dataBase->nodeVec),dst) == NULL))
    {
        addError(CRITICAL,"Source or destination node doesn’t exist in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    if (DBlinkTabSet(&(dataBase->linkTab),link,src,dst)<0)
    {
        addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
    ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[dst]->inNeighb),src));

    ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[src]->outNeighb)));
    ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[dst]->inNeighb)));

    ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,src+1));
    ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,dst+1));

    // Maximum non-null element
    dataBase->linkSrcVec.top = max(dataBase->linkSrcVec.top, id+1);
    dataBase->linkDstVec.top = dataBase->linkSrcVec.top;

    if (ret<0)
    {
        addError(CRITICAL,"Link addition uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks++;

    return ret;
}
```

4.12.3.3 int DBaddLSP (DataBase ∗ *dataBase*, DBLabelSwitchedPath ∗ *lsp*, LongList ∗ *preemptList*)

Definition at line 1736 of file database.c.

References addError(), ANDERROR, chooseReroutedLSPs(), LongVec_::cont, CRITICAL, DBgetLinkLSPs(), DBlinkTabGet, DBlspCopy(), DBlspDestroy(), DBlspListInsert(), DBlspNew(), DBlspVecGet, DBlspVecSet(), DBupdateLSOnSetup(), FALSE, GLOBAL_BACK, DBLabelSwitchedPath_::id, DataBase_::id, DBLink_::id, isValidLSPLink(), DataBase_::linkSrcVec, DataBase_::linkTab, LOCAL_BACK, longListEnd, longListInit, longListMerge(), longVecEnd(), longVecInit(), DBLink_::lspList, DataBase_::lspVec, NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, DBLabelSwitchedPath_::precedence, DBLabelSwitchedPath_::primPath, LongVec_::size, DBLink_::state, LongVec_::top, TRUE, DBLabelSwitchedPath_::type, and WARNING.

```
{
    DBLabelSwitchedPath *newLSP, *contentLSP=NULL;
    DBLSPList *lspList;
    int i,ret=0;
    DBLink *lnk=NULL;
    LongVec isProcessed;
    double rerouteGain[NB_OA];
    bool allowLSP=TRUE;
#if defined SIMULATOR
    LongList idList;
#elif defined AGENT
    int j;
    bool inPath=FALSE;
#endif

#if defined LINUX && defined TIME2
    struct timezone tz;
    struct timeval  t1,t2;
#endif

    if (dataBase == NULL || lsp==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

#if defined LINUX && defined TIME2
    gettimeofday(&t1, &tz);
#endif

    if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
    {
        addError(CRITICAL,"Unable to initialize vector of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(rerouteGain,0,NB_OA*sizeof(double));

    // Check if establishment is possible
#if defined SIMULATOR
    if (longListInit(&(idList),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize list of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    for (i=0;(i<lsp->path.top-1) && allowLSP;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);



        allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
                                            &(lnk->state),lsp,rerouteGain);
        if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        idList.top=0;
        if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
        {
            addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        if (longListMerge(&(idList),preemptList,preemptList)<0)
        {
            addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        isProcessed.cont[lnk->id] = 1;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;(i<lsp->primPath.top-1) && allowLSP;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[i],lsp->primPath.co
                                                 &(lnk->state),lsp,rerouteGain);
                if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==
                {
                    addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at li
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                idList.top=0;
                if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<
                {
                    addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                if (longListMerge(&(idList),preemptList,preemptList)<0)
                {
                    addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
    longListEnd(&(idList));
#elif defined AGENT
    for (i=0;(i<lsp->path.top-1) && (lsp->path.cont[i]!=dataBase->id);i++);

    if (i<lsp->path.top-1)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
                                             &(lnk->state),lsp,rerouteGain);
        if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
                     __FILE__,__LINE__);
            longVecEnd(&(isProcessed));
        }
        if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)<0)
        {
            addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                     __FILE__,__LINE__);
            longVecEnd(&(isProcessed));
            return -1;
        }
        isProcessed.cont[lnk->id] = 1;
        inPath=TRUE;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (j=0;(j<lsp->primPath.top-1) && (lsp->primPath.cont[j]!=dataBase->id);j++);

        if (j<lsp->primPath.top-1)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[j],
                             lsp->primPath.cont[j+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[j],lsp->primPath.cor
                                                     &(lnk->state),lsp,rerouteGain);
                if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==
                {
                    addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at li
                             __FILE__,__LINE__);
                    longVecEnd(&(isProcessed));
                }
                if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)
                {
                    addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                             __FILE__,__LINE__);
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                isProcessed.cont[lnk->id] = 1;
            }
            inPath=TRUE;
        }
    }
    if (!inPath)
    {
        addError(CRITICAL,"Agent not concerned by this LSP in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    if (!allowLSP)
    {
        addError(CRITICAL,"LSP refused by the predicate in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }


    if ((newLSP=DBlspNew())==NULL)
    {
        addError(CRITICAL,"Unable to create LSP in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (DBlspCopy(newLSP,lsp)<0)
    {
        addError(CRITICAL,"Unable to create a valid LSP copy in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(newLSP);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (DBlspVecSet(&(dataBase->lspVec),newLSP,newLSP->id)<0)
    {
        addError(CRITICAL,"Unable to insert LSP in the general LSP container in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(newLSP);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (newLSP->noContentionId>=0)
    {
        if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),newLSP->noContentionId))==NULL)
        {
            addError(WARNING,"Unable to get no contention LSP in %s at line %d",
                     __FILE__,__LINE__);
            newLSP->noContentionId=-1;
            // not critical enough to abort
        }
        else
        {
            contentLSP->noContentionId=newLSP->id;
        }
    }

    for (i=0;i<isProcessed.size;i++)
    {
        isProcessed.cont[i]=0;
    }


#if defined SIMULATOR
    // Add the LSP to each link list and update all the linkstates (only once !!!!!)
    for (i=0;i<newLSP->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
                         newLSP->path.cont[i+1]);
        ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
        ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->path.cont[i],
                                       newLSP->path.cont[i+1], &(lnk->state), newLSP));
        isProcessed.cont[lnk->id] = 1;
    }
    if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
    {
        for (i=0;i<newLSP->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
                             newLSP->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->primPath.cont[i],
                                               newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
#elif defined AGENT
    // Add the LSP to the link attached to the agent and update the linkstate
    for (i=0;i<newLSP->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
                         newLSP->path.cont[i+1]);
        ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));

        if (newLSP->path.cont[i] == dataBase->id)
        {
            ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->path.cont[i],
                                           newLSP->path.cont[i+1], &(lnk->state), newLSP));
            isProcessed.cont[lnk->id] = 1;
        }
    }
    if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
    {
        for (i=0;i<newLSP->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
                             newLSP->primPath.cont[i+1]);

            if (newLSP->primPath.cont[i] == dataBase->id)
            {
                if (isProcessed.cont[lnk->id] == 0)
                {
                    ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->primPath.cont[i],
                                                   newLSP->primPath.cont[i+1], &(lnk->state), newLSP)
                }
                break;
            }
        }
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    longVecEnd(&(isProcessed));

#if defined LINUX && defined TIME2
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time to add a new LSP : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif


    if (ret<0)
    {
        addError(CRITICAL,"LSP addition uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.12.3.4 int DBaddNode (DataBase ∗ *dataBase*, long *id*)

Definition at line 1523 of file database.c.

References addError(), CRITICAL, DBnodeDestroy(), DBnodeNew(), DBnodeVecSet(), DBNode_::id, DataBase_::nbNodes, and DataBase_::nodeVec.

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((node=DBnodeNew()) == NULL)
    {
        addError(CRITICAL,"Unable to create node in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    node->id=id;

    if (DBnodeVecSet(&(dataBase->nodeVec),node,id) < 0)
    {
        addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %c
                 __FILE__,__LINE__);
        DBnodeDestroy(node);
        return -1;
    }

    dataBase->nbNodes++;

    return 0;
}
```

4.12.3.5 int DBdestroy (DataBase ∗ *dataBase*)

Definition at line 1406 of file database.c.

References addError(), CRITICAL, DBlinkTabEnd(), DBlspVecEnd(), DBnodeVecEnd(), free, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, longVecEnd(), DataBase_::lspVec, and DataBase_::nodeVec.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBnodeVecEnd(&(dataBase->nodeVec));
    DBlspVecEnd(&(dataBase->lspVec));
    DBlinkTabEnd(&(dataBase->linkTab));
    longVecEnd(&(dataBase->linkSrcVec));
    longVecEnd(&(dataBase->linkDstVec));

    free(dataBase);

1423     return 0;
1424 }
```

4.12.3.6 int DBevalLSOnRemove (DataBase * *dataBase*, long *src*, long *dst*, DBLinkState * *newLS*, DBLinkState * *oldLS*, LSPRequest * *req*)

Definition at line 1316 of file database.c.

References evalLS(), and REMOVE.

Referenced by computeBackup().

```
1317 {
1318     return evalLS(dataBase, src, dst, newLS, oldLS, req, REMOVE);
1319 }
```

4.12.3.7 int DBevalLSOnSetup (DataBase * *dataBase*, long *src*, long *dst*, DBLinkState * *newLS*, DBLinkState * *oldLS*, LSPRequest * *req*)

Definition at line 1310 of file database.c.

References evalLS(), and SETUP.

Referenced by computeBackup(), computeCost(), and isValidRequestLink().

```
1311 {
1312     return evalLS(dataBase, src, dst, newLS, oldLS, req, SETUP);
1313 }
```

4.12.3.8 long DBgetID (DataBase * *dataBase*)

Definition at line 1426 of file database.c.

References addError(), CRITICAL, and DataBase_::id.

```
1427 {
1428     if (dataBase == NULL)
1429     {
1430         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1431                  __FILE__,__LINE__);
1432         return -1;
1433     }
1434
1435     return dataBase->id;
1436 }
```

4.12.3.9 long DBgetLinkDst (DataBase * *dataBase*, long *id*)

Definition at line 1478 of file database.c.

References addError(), CRITICAL, DataBase_::linkDstVec, and longVecGet().

Referenced by computeBackup().

```
{
    long ret;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecGet(&(dataBase->linkDstVec),id,&ret)<0)
    {
        addError(CRITICAL,"Inexistent link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return (ret-1);
}
```

4.12.3.10 long DBgetLinkID (DataBase ∗ *dataBase*, long *src*, long *dst*)

Definition at line 1438 of file database.c.

References addError(), CRITICAL, DBlinkTabGet, DBLink_::id, and DataBase_::linkTab.

Referenced by computeBackup(), computeCost(), DBprintDB(), DBremoveLink(), and updateLS().

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL || src < 0 || dst < 0)
    {
        addError(CRITICAL,"Bad argument (NULL or negative value) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst))==NULL)
    {
        return -1;
    }

    return lnk->id;
}
```

4.12.3.11 DBLSPList∗ DBgetLinkLSPs (DataBase ∗ *dataBase*, long *src*, long *dst*)

Definition at line 2198 of file database.c.

References addError(), CRITICAL, DBlinkTabGet, DataBase_::linkTab, and DBLink_::lspList.

Referenced by DBaddLSP().

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return NULL;
    }

    return &(lnk->lspList);
}
```

4.12.3.12 long DBgetLinkSrc (DataBase ∗ *dataBase*, long *id*)

Definition at line 1457 of file database.c.

References addError(), CRITICAL, DataBase_::linkSrcVec, and longVecGet().

Referenced by computeBackup().

```
{
    long ret;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecGet(&(dataBase->linkSrcVec),id,&ret)<0)
    {
        addError(CRITICAL,"Inexistent link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return (ret-1);
}
```

4.12.3.13 DBLinkState∗ DBgetLinkState (DataBase ∗ *dataBase*, long *src*, long *dst*)

Definition at line 2219 of file database.c.

References addError(), CRITICAL, DBlinkTabGet, DataBase_::linkTab, and DBLink_::state.

Referenced by computeBackup(), and fillTopo().

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
```

```
2232         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2233                  src,dst,__FILE__,__LINE__);
2234         return NULL;
2235     }
2236 
2237     return &(lnk->state);
2238 }
```

4.12.3.14 DBLabelSwitchedPath∗ DBgetLSP (DataBase ∗ *dataBase*, long *id*)

Definition at line 2185 of file database.c.

References addError(), CRITICAL, DBlspVecGet, and DataBase_::lspVec.

Referenced by computeBackup(), evalLS(), and updateLS().

```
2186 {
2187     if (dataBase == NULL)
2188     {
2189         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2190                  __FILE__,__LINE__);
2191         return NULL;
2192     }
2193 
2194     return DBlspVecGet(&(dataBase->lspVec), id);
2195 }
```

4.12.3.15 long DBgetMaxNodeID (DataBase ∗ *dataBase*)

Definition at line 1511 of file database.c.

References addError(), CRITICAL, DataBase_::nodeVec, and DBNodeVec_::top.

Referenced by fillTopo().

```
1512 {
1513     if (dataBase == NULL)
1514     {
1515         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1516                  __FILE__,__LINE__);
1517         return -1;
1518     }
1519 
1520     return dataBase->nodeVec.top-1;
1521 }
```

4.12.3.16 long DBgetNbLinks (DataBase ∗ *dataBase*)

Definition at line 1600 of file database.c.

References addError(), CRITICAL, and DataBase_::nbLinks.

Referenced by fillTopo().

```
1601 {
1602     if (dataBase == NULL)
1603     {
1604         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nbLinks;
}
```

4.12.3.17 long DBgetNbNodes (DataBase * *dataBase*)

Definition at line 1499 of file database.c.

References addError(), CRITICAL, and DataBase_::nbNodes.

Referenced by fillTopo().

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nbNodes;
}
```

4.12.3.18 LongList* DBgetNodeInNeighb (DataBase * *dataBase*, long *id*)

Definition at line 2269 of file database.c.

References addError(), CRITICAL, DBnodeVecGet, DBNode_::inNeighb, and DataBase_::nodeVec.

Referenced by computeBackup(), and fillTopo().

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
    {
        addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return NULL;
    }

    return (&(node->inNeighb));
}
```

4.12.3.19 LongList* DBgetNodeOutNeighb (DataBase * *dataBase*, long *id*)

Definition at line 2291 of file database.c.

References addError(), CRITICAL, DBnodeVecGet, DataBase_::nodeVec, and DBNode_::outNeighb.

Referenced by computeBackup(), and fillTopo().

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
    {
        addError(CRITICAL,"Node don’t exist (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return NULL;
    }

    return (&(node->outNeighb));
}
```

4.12.3.20 int DBlinkStateCopy (DBLinkState ∗ *dst*, DBLinkState ∗ *src*)

Definition at line 660 of file database.c.

References addError(), ANDERROR, DBLinkState_::bbw, DBLinkState_::cap, DBLinkState_::color, CRITICAL, dblVecCopy(), DBLinkState_::fbw, NB_OA, NB_PREEMPTION, DBLinkState_::pbw, DBLinkState_::rbw, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeBackup(), DBaddLink(), DBsetLinkState(), and evalLS().

```
{
    int i,j,ret=0;

    if (dst == NULL || src == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    dst->color=src->color;
    memcpy(&(dst->cap),&(src->cap),NB_OA * sizeof(double));
    memcpy(&(dst->rbw),&(src->rbw),NB_OA * NB_PREEMPTION * sizeof(double));
    memcpy(&(dst->pbw),&(src->pbw),NB_OA * NB_PREEMPTION * sizeof(double));

    for (i=0;(i<NB_OA && ret>=0);i++)
        for (j=0;(j<NB_PREEMPTION && ret>=0);j++)
        {
            ANDERROR(ret,dblVecCopy(&(dst->bbw[i][j]),&(src->bbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->remoteBbw[i][j]),&(src->remoteBbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->fbw[i][j]),&(src->fbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->remoteFbw[i][j]),&(src->remoteFbw[i][j])));
        }

    if (ret<0)
    {
        addError(CRITICAL,"Link state copy uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.12.3.21 int DBlinkStateDestroy (DBLinkState ∗ *ls*)

Definition at line 613 of file database.c.

References addError(), DBLinkState_::bbw, CRITICAL, dblVecEnd(), DBLinkState_::fbw, free, NB_OA, NB_PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeBackup().

```
{
    int i,j;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            dblVecEnd(&(ls->bbw[i][j]));
            dblVecEnd(&(ls->remoteBbw[i][j]));
            dblVecEnd(&(ls->fbw[i][j]));
            dblVecEnd(&(ls->remoteFbw[i][j]));
        }
    free(ls);

    return 0;
}
```

4.12.3.22 int DBlinkStateEnd (DBLinkState ∗ *ls*)

Definition at line 637 of file database.c.

References addError(), DBLinkState_::bbw, CRITICAL, dblVecEnd(), DBLinkState_::fbw, NB_OA, NB_PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeCost(), DBlinkDestroy(), DBlinkEnd(), DBlinkInit(), DBlinkNew(), and isValidRequestLink().

```
{
    int i,j;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            dblVecEnd(&(ls->bbw[i][j]));
            dblVecEnd(&(ls->remoteBbw[i][j]));
            dblVecEnd(&(ls->fbw[i][j]));
            dblVecEnd(&(ls->remoteFbw[i][j]));
        }

    return 0;
}
```

4.12.3.23 int DBlinkStateInit (DBLinkState ∗ *ls*)

Definition at line 530 of file database.c.

References addError(), DBLinkState_::bbw, CRITICAL, dblVecEnd(), dblVecInit(), DBLinkState_::fbw, NB_OA, NB_PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeCost(), DBlinkInit(), DBlinkNew(), and isValidRequestLink().

```
{
    int i,j,k,l;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(ls, 0, sizeof(DBLinkState));

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
            {
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                dblVecEnd(&(ls->fbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
        }

    return 0;
}
```

4.12.3.24 DBLinkState∗ DBlinkStateNew ()

Definition at line 444 of file database.c.

References addError(), DBLinkState_::bbw, calloc, CRITICAL, dblVecEnd(), dblVecInit(), DBLinkState_::fbw, free, NB_OA, NB_PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

```
{
    DBLinkState* ls;
    int i,j,k,l;

    if ((ls=calloc(1,sizeof(DBLinkState)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
            {
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                 __FILE__,__LINE__);
                return NULL;
            }
            else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                for (k=i;k>=0;k--)
```

```
                        for (l=j-1;l>=0;l--)
                        {
                            dblVecEnd(&(ls->bbw[k][l]));
                            dblVecEnd(&(ls->remoteBbw[k][l]));
                            dblVecEnd(&(ls->fbw[k][l]));
                            dblVecEnd(&(ls->remoteFbw[k][l]));
                        }
                    free(ls);
                    addError(CRITICAL,"Unable to create link state in %s at line %d",
                             __FILE__,__LINE__);
                    return NULL;
                }
                else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
                {
                    dblVecEnd(&(ls->bbw[i][j]));
                    dblVecEnd(&(ls->remoteBbw[i][j]));
                    for (k=i;k>=0;k--)
                        for (l=j-1;l>=0;l--)
                        {
                            dblVecEnd(&(ls->bbw[k][l]));
                            dblVecEnd(&(ls->remoteBbw[k][l]));
                            dblVecEnd(&(ls->fbw[k][l]));
                            dblVecEnd(&(ls->remoteFbw[k][l]));
                        }
                    free(ls);
                    addError(CRITICAL,"Unable to create link state in %s at line %d",
                             __FILE__,__LINE__);
                    return NULL;
                }
                else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
                {
                    dblVecEnd(&(ls->bbw[i][j]));
                    dblVecEnd(&(ls->remoteBbw[i][j]));
                    dblVecEnd(&(ls->fbw[i][j]));
                    for (k=i;k>=0;k--)
                        for (l=j-1;l>=0;l--)
                        {
                            dblVecEnd(&(ls->bbw[k][l]));
                            dblVecEnd(&(ls->remoteBbw[k][l]));
                            dblVecEnd(&(ls->fbw[k][l]));
                            dblVecEnd(&(ls->remoteFbw[k][l]));
                        }
                    free(ls);
                    addError(CRITICAL,"Unable to create link state in %s at line %d",
                             __FILE__,__LINE__);
                    return NULL;
                }
            }

    return ls;
}
```

4.12.3.25 int DBlspCompare (const DBLabelSwitchedPath ∗ *LSPa*, const DBLabelSwitchedPath ∗ *LSPb*)

Definition at line 357 of file database.c.

References DBLabelSwitchedPath_::bw, DBLabelSwitchedPath_::id, and DBLabelSwitchedPath_::precedence.

Referenced by DBlspListInsert(), and DBlspListRemove().

```
{
    if (LSPa->precedence > LSPb->precedence)
```

```
        return 1;
    else if (LSPa->precedence < LSPb->precedence)
        return -1;
    else if (LSPa->bw[0] > LSPb->bw[0])
        return 1;
    else if (LSPa->bw[0] < LSPb->bw[0])
        return -1;
    else
    {
        if (LSPa->id < LSPb->id)
            return 1;
        else if (LSPa->id > LSPb->id)
            return -1;
    }

    return 0;
}
```

4.12.3.26 int DBlspCopy (DBLabelSwitchedPath ∗ *dst*, DBLabelSwitchedPath ∗ *src*)

Definition at line 157 of file database.c.

References addError(), ANDERROR, DBLabelSwitchedPath_::backLSPIDs, DBLabelSwitchedPath_::bw, CRITICAL, DBLabelSwitchedPath_::forbidLinks, DBLabelSwitchedPath_::id, longListCopy, NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, DBLabelSwitchedPath_::precedence, DBLabelSwitchedPath_::primID, DBLabelSwitchedPath_::primPath, and DBLabelSwitchedPath_::type.

Referenced by DBaddLSP().

```
{
    int ret=0;

    if (dst == NULL || src==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    dst->id=src->id;
    dst->precedence=src->precedence;
    memcpy(dst->bw,src->bw, NB_OA * sizeof(double));
    dst->noContentionId = src->noContentionId;
    ANDERROR(ret,longListCopy(&(dst->forbidLinks),&(src->forbidLinks)));
    ANDERROR(ret,longListCopy(&(dst->path),&(src->path)));
    dst->type=src->type;
    dst->primID=src->primID;
    ANDERROR(ret,longListCopy(&(dst->primPath),&(src->primPath)));
    ANDERROR(ret,longListCopy(&(dst->backLSPIDs),&(src->backLSPIDs)));

    if (ret<0)
    {
        addError(CRITICAL,"Label switched path copy uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.12.3.27 int DBlspDestroy (DBLabelSwitchedPath ∗ *lsp*)

Definition at line 122 of file database.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, CRITICAL, DBLabelSwitchedPath_::forbidLinks, free, longListEnd, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

Referenced by DBaddLSP(), DBlspVecDestroy(), DBlspVecEnd(), DBlspVecResize(), and evalLS().

```
123 {
124     if (lsp == NULL)
125     {
126         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
127                  __FILE__,__LINE__);
128         return -1;
129     }
130
131     longListEnd(&(lsp->backLSPIDs));
132     longListEnd(&(lsp->primPath));
133     longListEnd(&(lsp->path));
134     longListEnd(&(lsp->forbidLinks));
135     free(lsp);
136
137     return 0;
138 }
```

4.12.3.28 int DBlspEnd (DBLabelSwitchedPath ∗ *lsp*)

Definition at line 140 of file database.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, CRITICAL, DBLabelSwitchedPath_::forbidLinks, longListEnd, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

```
141 {
142     if (lsp == NULL)
143     {
144         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
145                  __FILE__,__LINE__);
146         return -1;
147     }
148
149     longListEnd(&(lsp->backLSPIDs));
150     longListEnd(&(lsp->primPath));
151     longListEnd(&(lsp->path));
152     longListEnd(&(lsp->forbidLinks));
153
154     return 0;
155 }
```

4.12.3.29 int DBlspInit (DBLabelSwitchedPath ∗ *lsp*)

Definition at line 73 of file database.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, DBLabelSwitchedPath_::bw, CRITICAL, DBLabelSwitchedPath_::forbidLinks, longListEnd, longListInit, NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

```
74 {
75     if (lsp == NULL)
```

```
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->forbidLinks),-1)<0)
    {
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->path),-1)<0)
    {
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->primPath),-1)<0)
    {
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->backLSPIDs),-1)<0)
    {
        longListEnd(&(lsp->primPath));
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(lsp->bw, 0, NB_OA * sizeof(double));
    lsp->noContentionId=-1;   //very important

    return 0;
}
```

4.12.3.30 int DBlspListDestroy (DBLSPList ∗ *list*)

Definition at line 251 of file database.c.

References addError(), DBLSPList_::cont, CRITICAL, and free.

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(list->cont);
    free(list);

    return 0;
}
```

4.12.3.31 int DBlspListEnd (DBLSPList * *list*)

Definition at line 266 of file database.c.

References addError(), DBLSPList_::cont, CRITICAL, free, DBLSPList_::size, and DBLSPList_::top.

Referenced by DBlinkDestroy(), and DBlinkEnd().

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(list->cont);
    list->cont = NULL;
    list->size = 0;
    list->top = 0;

    return 0;
}
```

4.12.3.32 int DBlspListInit (DBLSPList * *list*, long *size*)

Definition at line 223 of file database.c.

References addError(), calloc, DBLSPList_::cont, CRITICAL, LSPLIST_INITSIZE, DBLSPList_::size, and DBLSPList_::top.

Referenced by DBlinkInit(), and DBlinkNew().

```
{
    void* ptr=NULL;

    if (list == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LSPLIST_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    list->size = size;
    list->top = 0;
    list->cont = ptr;

    return 0;
}
```

4.12.3.33 int DBlspListInsert (DBLSPList ∗ *list*, DBLabelSwitchedPath ∗ *lsp*)

Definition at line 283 of file database.c.

References addError(), DBLSPList_::cont, CRITICAL, DBlspCompare(), realloc, DBLSPList_::size, and DBLSPList_::top.

Referenced by DBaddLSP().

```
{
    int a,b;
    void *ptr=NULL;

    if (list == NULL || list->cont == NULL || lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // check the capacity of the list
    if (list->top >= list->size)
    {
        if ((ptr = realloc(list->cont, list->size
                          * 2 * sizeof(DBLabelSwitchedPath*))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            list->cont=ptr;
            list->size*=2;
        }
    }

    // find the position in the list (to keep it sorted)
    a = 0;
    b = list->top-1;

    // empty list or after the last elem
    if (list->top == 0 || DBlspCompare(list->cont[b], lsp) >= 0)
    {
        list->cont[list->top++] = lsp;
        return (list->top-1);
    }

    // before the first elem
    if (DBlspCompare(lsp, list->cont[a]) >= 0)
    {
        memmove(list->cont+1, list->cont, (list->top)*sizeof(void*));
        list->cont[0] = lsp;
        list->top++;
        return 0;
    }

    // now the insert position is inside ]a,b[
    while (b - a > 1)
    {
        int mid = (a + b)/2;
        int ret = DBlspCompare(lsp, list->cont[mid]);

        if (ret == 1)
            b = mid;
        else if (ret == -1)
            a = mid;
        else // if (ret == 0)
        {
            a = mid;
            b = mid;
        }
    }

    // now insert before b
    memmove(list->cont+b+1, list->cont+b, (list->top - b)*sizeof(void*));
    list->cont[b] = lsp;
    list->top++;

    return b;
}
```

4.12.3.34 DBLSPList* DBlspListNew (long *size*)

Definition at line 193 of file database.c.

References addError(), calloc, DBLSPList_::cont, CRITICAL, free, LSPLIST_INITSIZE, DBLSPList_::size, and DBLSPList_::top.

```
{
    DBLSPList *list=NULL;
    void* ptr=NULL;

    if ((list = calloc(1,sizeof(DBLSPList))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LSPLIST_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(list);
        return NULL;
    }

    list->size = size;
    list->top = 0;
    list->cont = ptr;

    return list;
}
```

4.12.3.35 int DBlspListRemove (DBLSPList * *list*, DBLabelSwitchedPath * *lsp*)

Definition at line 378 of file database.c.

References addError(), DBLSPList_::cont, CRITICAL, DBlspCompare(), DBLSPList_::top, and WARNING.

```
{
    int a,b,index;

    if (list == NULL || list->cont == NULL || lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // find the position in the list
    a = 0;
    b = list->top-1;

    // empty list
    if (list->top == 0)
    {
        addError(WARNING,"Removing inexistent LSP in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    while (b - a > 1)
    {
        int mid = (a + b)/2;
        int ret = DBlspCompare(lsp, list->cont[mid]);

        if (ret == 1)
            b = mid;
        else if (ret == -1)
            a = mid;
        else // if (ret == 0)
        {
            a = mid;
            b = mid;
        }
    }

    if (DBlspCompare(lsp, list->cont[a]) == 0)
    {
        index = a;
    }
    else if (DBlspCompare(lsp, list->cont[b]) == 0)
    {
        index = b;
    }
    else // not found
    {
        addError(WARNING,"Removing inexistent LSP in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // now delete index
    memmove(list->cont + index, list->cont + index + 1, (list->top - index -1)*sizeof(void*));
    list->top--;

    return 0;
}
```

4.12.3.36 DBLabelSwitchedPath∗ DBlspNew ()

Definition at line 19 of file database.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, calloc, CRITICAL, DBLabelSwitchedPath_::forbidLinks, free, longListEnd, longListInit, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

```
{
    DBLabelSwitchedPath* lsp;

    if ((lsp=calloc(1,sizeof(DBLabelSwitchedPath)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->forbidLinks),-1)<0)
    {
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->path),-1)<0)
    {
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->primPath),-1)<0)
    {
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->backLSPIDs),-1)<0)
    {
        longListEnd(&(lsp->primPath));
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    lsp->noContentionId=-1; //very important

    return lsp;
}
```

4.12.3.37 DataBase∗ DBnew (long *ID*)

Definition at line 1337 of file database.c.

References addError(), calloc, CRITICAL, DBlinkTabEnd(), DBlinkTabInit(), DBlspVecEnd(), DBlspVecInit(), DBnodeVecEnd(), DBnodeVecInit(), free, DataBase_::id, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, LINKTAB_INITSIZE, longVecEnd(), longVecInit(), DataBase_::lspVec, DataBase_::nbLinks, DataBase_::nbNodes, and DataBase_::nodeVec.

```
{
    DataBase *dataBase=NULL;

    if ((dataBase=calloc(1,sizeof(DataBase)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    dataBase->id=ID;

    if (DBnodeVecInit(&(dataBase->nodeVec),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general node container in %s at line %d",
                 __FILE__,__LINE__);
        free(dataBase);
        return NULL;
    }

    if (DBlspVecInit(&(dataBase->lspVec),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general LSP container in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        free(dataBase);
        return NULL;
    }

    if (DBlinkTabInit(&(dataBase->linkTab),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general link container in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        free(dataBase);
        return NULL;
    }

    if (longVecInit(&(dataBase->linkSrcVec),LINKTAB_INITSIZE)<0)
    {
        addError(CRITICAL,"Unable to initialize the link id-src translater in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        DBlinkTabEnd(&(dataBase->linkTab));
        free(dataBase);
        return NULL;
    }

    if (longVecInit(&(dataBase->linkDstVec),LINKTAB_INITSIZE)<0)
    {
        addError(CRITICAL,"Unable to initialize the link id-dst translater in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        DBlinkTabEnd(&(dataBase->linkTab));
        longVecEnd(&(dataBase->linkSrcVec));
        free(dataBase);
        return NULL;
    }

    dataBase->nbNodes=0;
    dataBase->nbLinks=0;

    return dataBase;
}
```

4.12.3.38 void DBprintDB (DataBase ∗ *db*)

Definition at line 2313 of file database.c.

References DBLinkTab_::cont, DBNodeVec_::cont, DBgetLinkID(), DBprintLink(), DBprintNode(), DataBase_::linkTab, DataBase_::nodeVec, DBLinkTab_::size, and DBNodeVec_::size.

```
{
    long i,j;

    printf("Printing info about nodes ...\n");
    printf("----------------------------\n");

    for (i=0; i<db->nodeVec.size; i++)
    {
        if (db->nodeVec.cont[i])
        {
            printf("Node id : %ld\n", i);
            printf("------------\n");
            DBprintNode(db->nodeVec.cont[i]);
        }
    }

    printf("\nPrinting info about links ...\n");
    printf("----------------------------\n");

    for (i=0; i<db->linkTab.size; i++)
        for (j=0; j<db->linkTab.size; j++)
        {
            if (db->linkTab.cont[i][j])
            {
                printf("Link %ld-%ld (id = %ld)\n", i, j, DBgetLinkID(db, i, j));
                printf("----------------------\n");

                DBprintLink(db->linkTab.cont[i][j]);

            }
        }
}
```

4.12.3.39 int DBremoveLink (DataBase ∗ *dataBase*, long *src*, long *dst*)

Definition at line 1692 of file database.c.

References addError(), ANDERROR, LongVec_::cont, DBNodeVec_::cont, CRITICAL, DBgetLinkID(), DBlinkTabGet, DBlinkTabRemove(), DBnodeVecGet, DBNode_::inNeighb, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, longListRemove(), longVecSet(), DataBase_::nbLinks, DataBase_::nodeVec, DBNode_::outNeighb, and LongVec_::top.

Referenced by DBremoveNode().

```
{
    int id,ret=0;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((DBnodeVecGet(&(dataBase->nodeVec),src)==NULL) ||
        (DBnodeVecGet(&(dataBase->nodeVec),dst)==NULL) ||
        (DBlinkTabGet(&(dataBase->linkTab),src,dst)==NULL))
    {
        addError(CRITICAL,"Link doesn’t exist or database unconsistancy in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
    ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[dst]->inNeighb),src));

    ANDERROR(ret,DBlinkTabRemove(&(dataBase->linkTab),src,dst));

    id=DBgetLinkID(dataBase,src,dst);
    ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,0));
    ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,0));

    while (dataBase->linkSrcVec.cont[dataBase->linkSrcVec.top-1] == 0)
        dataBase->linkSrcVec.top--;

    if (ret<0)
    {
        addError(CRITICAL,"Link removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks--;

    return ret;
}
```

4.12.3.40 int DBremoveLSP (DataBase ∗ *dataBase*, long *id*)

Definition at line 2065 of file database.c.

References addError(), ANDERROR, and DBlinkTabGet.

```
{
    DBLabelSwitchedPath *lsp=NULL, *contentLSP=NULL;
    int i,ret=0;
    DBLink *lnk=NULL;
    LongVec isProcessed;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lsp = DBlspVecGet(&(dataBase->lspVec), id)) == NULL)
    {
        addError(CRITICAL,"Trying to remove inexistent LSP (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return -1;
    }

    if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
    {
        addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

```

```
#if defined SIMULATOR
    // Remove the LSP from each link list and update all the linkstates
    for (i=0;i<lsp->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
        ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->path.cont[i],
                                        lsp->path.cont[i+1], &(lnk->state), lsp));
        isProcessed.cont[lnk->id] = 1;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;i<lsp->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->primPath.cont[i],
                                                lsp->primPath.cont[i+1], &(lnk->state), lsp));
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
#elif defined AGENT
    // Remove the LSP to the link attached to the agent and update the linkstate
    for (i=0;i<lsp->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));

        if (lsp->path.cont[i] == dataBase->id)
        {
            ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->path.cont[i],
                                            lsp->path.cont[i+1], &(lnk->state), lsp));
            isProcessed.cont[lnk->id] = 1;
        }
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;i<lsp->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);

            if (lsp->primPath.cont[i] == dataBase->id)
            {
                if (isProcessed.cont[lnk->id] == 0)
                {
                    ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->primPath.cont[i],
                                                    lsp->primPath.cont[i+1], &(lnk->state), lsp));
                }
                break;
            }
        }
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    longVecEnd(&(isProcessed));

    // remove the lsp from the global list
    ANDERROR(ret,DBlspVecRemove(&(dataBase->lspVec), id));

    if (lsp->noContentionId>=0)
    {
        if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),lsp->noContentionId))==NULL)
        {
            addError(WARNING,"Unable to get no contention LSP in %s at line %d",
                     __FILE__,__LINE__);
            // not critical enough to abort
        }
        contentLSP->noContentionId=-1;
    }

    // free the lsp
    DBlspDestroy(lsp);

    if (ret<0)
    {
        addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.12.3.41 int DBremoveNode (DataBase * *dataBase*, long *id*)

Definition at line 1556 of file database.c.

References addError(), ANDERROR, LongVec_::cont, CRITICAL, DBnodeVecGet, DBnodeVecRemove(), DBremoveLink(), DBNode_::inNeighb, DataBase_::nbLinks, DataBase_::nodeVec, DBNode_::outNeighb, and LongVec_::top.

```
{
    DBNode *node=NULL;
    int ret=0;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec),id)) == NULL)
    {
        addError(CRITICAL,"Trying to remove an inexistent node in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // remember that DBremoveLink will update the neighbour list
    while(node->inNeighb.top > 0)
    {
        ANDERROR(ret,DBremoveLink(dataBase,node->inNeighb.cont[node->inNeighb.top-1],id));
    }

    // remember that DBremoveLink will update the neighbour list
    while(node->outNeighb.top > 0)
    {
        ANDERROR(ret,DBremoveLink(dataBase,id,node->outNeighb.cont[node->outNeighb.top-1]));
    }

    ANDERROR(ret,DBnodeVecRemove(&(dataBase->nodeVec),id));

    if (ret<0)
    {
        addError(CRITICAL,"Node removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks--;

    return ret;
}
```

4.12.3.42 int DBsetLinkState (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *newLS*)

Definition at line 2240 of file database.c.

References addError(), CRITICAL, DBlinkStateCopy(), DBlinkTabGet, DataBase_::linkTab, and DBLink_::state.

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL || newLS == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return -1;
    }

    if (DBlinkStateCopy(&(lnk->state), newLS)<0)
    {
        addError(CRITICAL,"Impossible to set linkstate on link (src = %ld, dst = %ld) in %s at line %
                 src,dst,__FILE__,__LINE__);
        return -1;
    }

    return 0;
}
```

4.12.3.43 int DBupdateLSOnRemove (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *ls*, DBLabelSwitchedPath ∗ *lsp*)

Definition at line 1326 of file database.c.

References REMOVE, and updateLS().

```
{
    return updateLS(dataBase, src, dst, ls, lsp, REMOVE);
}
```

4.12.3.44 int DBupdateLSOnSetup (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *ls*, DBLabelSwitchedPath ∗ *lsp*)

Definition at line 1321 of file database.c.

References SETUP, and updateLS().

Referenced by DBaddLSP().

```
{
    return updateLS(dataBase, src, dst, ls, lsp, SETUP);
}
```

4.12.3.45 int evalLS (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *newLS*, DBLinkState ∗ *oldLS*, LSPRequest ∗ *req*, operation *op*)

Definition at line 838 of file database.c.

References addError(), LSPRequest_::bw, DBLabelSwitchedPath_::bw, CRITICAL, DBgetLSP(), DBlinkStateCopy(), DBlspDestroy(), DBlspNew(), LSPRequest_::forbidLinks, DBLabelSwitchedPath_::forbidLinks, GLOBAL_BACK, LSPrerouteInfo_::id, LSPRequest_::id, DBLabelSwitchedPath_::id, LOCAL_BACK, longListCopy, NB_OA, DBLabelSwitchedPath_::noContentionId, LSPRequest_::path, DBLabelSwitchedPath_::path, LSPRequest_::precedence, DBLabelSwitchedPath_::precedence, PRIM, LSPRequest_::primID, DBLabelSwitchedPath_::primID, DBLabelSwitchedPath_::primPath, LSPRequest_::rerouteInfo, LongVec_::top, DBLabelSwitchedPath_::type, LSPRequest_::type, and updateLS().

Referenced by DBevalLSOnRemove(), and DBevalLSOnSetup().

```
{




    DBLabelSwitchedPath* lsp, *primLSP;
    int ret;

    // check the arguments
    if ((dataBase==NULL) || (newLS==NULL) || (oldLS==NULL) || (req==NULL))
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // duplicate the LS
    if (newLS != oldLS && DBlinkStateCopy(newLS, oldLS) < 0)
    {
        addError(CRITICAL,"Impossible to duplicate the linkState in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // now build a false LSP satisfying the request ....
    lsp = DBlspNew();
    lsp->id = req->id;
    lsp->precedence = req->precedence;
    memcpy(lsp->bw, req->bw, NB_OA * sizeof(double));
    longListCopy(&(lsp->forbidLinks), &(req->forbidLinks));

    if (req->rerouteInfo.id >= 0)
    {
        lsp->noContentionId = req->rerouteInfo.id;
    }

    switch(req->type)
    {
        case PRIM:
            lsp->type = PRIM;
            lsp->primID = -1;
            break;

        case GLOBAL_BACK:
        case LOCAL_BACK:
            lsp->type = req->type;
            lsp->primID = req->primID;

            // look up the primary path ....
            if ((primLSP = DBgetLSP(dataBase, lsp->primID)) == NULL)
            {
                addError(CRITICAL,"Impossible to determine the primary path in %s at line %d",
                 __FILE__,__LINE__);
                DBlspDestroy(lsp);
                return -1;
            }

            longListCopy(&(lsp->primPath), &(primLSP->path));

            break;

        default:
            addError(CRITICAL,"Unknown request type (NULL) in %s at line %d",
                 __FILE__,__LINE__);
            DBlspDestroy(lsp);
            return -1;
    }

    if (req->path.top < 2)
    {
        addError(CRITICAL,"Wrong path in request in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(lsp);
        return -1;
    }


   if (longListCopy(&(lsp->path), &(req->path)) < 0)
    {
        addError(CRITICAL,"Impossible to duplicate path in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(lsp);
        return -1;
    }


    ret = updateLS(dataBase, src, dst, newLS, lsp, op);

    // clean up ....
    DBlspDestroy(lsp);

    return ret;
}
```

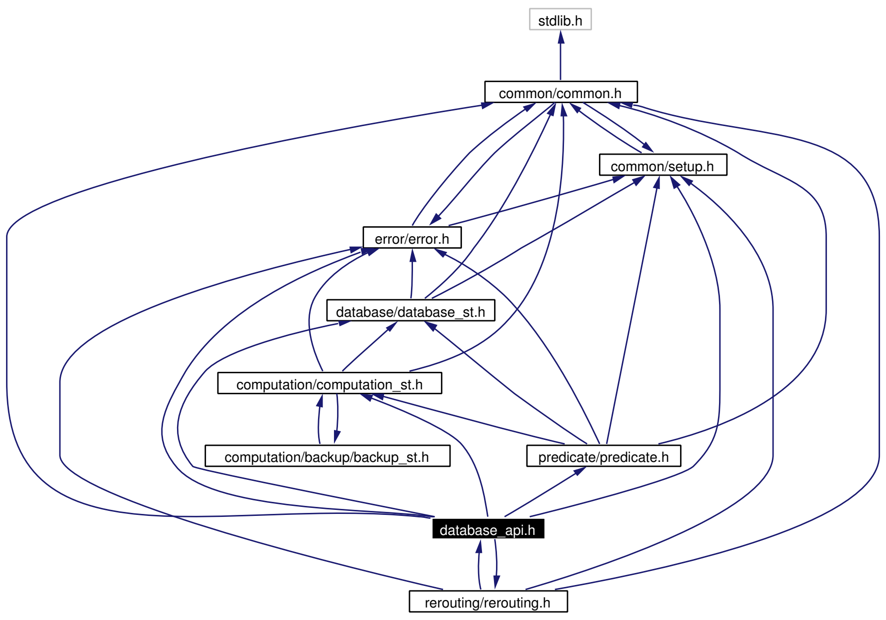
4.12.3.46 int updateLS (DataBase *, long, long, DBLinkState *, DBLabelSwitchedPath *, operation)

Referenced by DBupdateLSOnRemove(), DBupdateLSOnSetup(), and evalLS().

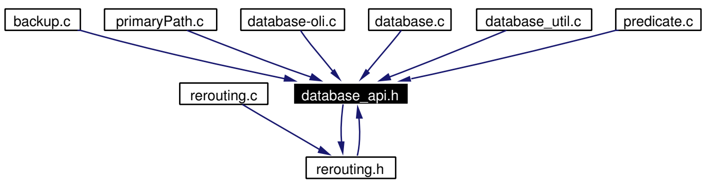
4.13 database_api.h File Reference

```
#include "common/common.h"
#include "common/setup.h"
#include "error/error.h"
#include "database/database_st.h"
#include "computation/computation_st.h"
#include "predicate/predicate.h"
#include "rerouting/rerouting.h"
```

Include dependency graph for database_api.h:



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



Functions

- DBLabelSwitchedPath ∗ DBlspNew ()
- int DBlspInit (DBLabelSwitchedPath ∗)
- int DBlspDestroy (DBLabelSwitchedPath ∗)
- int DBlspEnd (DBLabelSwitchedPath ∗)

- int DBlspCopy (DBLabelSwitchedPath ∗, DBLabelSwitchedPath ∗)
- DBLSPList ∗ DBlspListNew (long)
- int DBlspListInit (DBLSPList ∗, long)
- int DBlspListDestroy (DBLSPList ∗)
- int DBlspListEnd (DBLSPList ∗)
- int DBlspListInsert (DBLSPList ∗, DBLabelSwitchedPath ∗)
- int DBlspListRemove (DBLSPList ∗, DBLabelSwitchedPath ∗)
- int DBlspCompare (const DBLabelSwitchedPath ∗, const DBLabelSwitchedPath ∗)
- DBLinkState ∗ DBlinkStateNew ()
- int DBlinkStateInit (DBLinkState ∗)
- int DBlinkStateDestroy (DBLinkState ∗)
- int DBlinkStateEnd (DBLinkState ∗)
- int DBlinkStateCopy (DBLinkState ∗, DBLinkState ∗)
- int DBevalLSOnSetup (DataBase ∗, long, long, DBLinkState ∗, DBLinkState ∗, LSPRequest ∗)
- int DBevalLSOnRemove (DataBase ∗, long, long, DBLinkState ∗, DBLinkState ∗, LSPRequest ∗)
- int DBupdateLSOnSetup (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗)
- int DBupdateLSOnRemove (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗)
- DataBase ∗ DBnew (long)
- int DBdestroy (DataBase ∗)
- long DBgetID (DataBase ∗)
- long DBgetLinkID (DataBase ∗, long, long)
- long DBgetLinkSrc (DataBase ∗, long)
- long DBgetLinkDst (DataBase ∗, long)
- long DBgetNbNodes (DataBase ∗)
- long DBgetMaxNodeID (DataBase ∗)
- int DBaddNode (DataBase ∗, long)
- int DBremoveNode (DataBase ∗, long)
- long DBgetNbLinks (DataBase ∗)
- int DBaddLink (DataBase ∗, long, long, long, DBLinkState ∗)
- int DBremoveLink (DataBase ∗, long, long)
- int DBaddLSP (DataBase ∗, DBLabelSwitchedPath ∗, LongList ∗)
- int DBremoveLSP (DataBase ∗, long)
- DBLabelSwitchedPath ∗ DBgetLSP (DataBase ∗, long)
- DBLSPList ∗ DBgetLinkLSPs (DataBase ∗, long, long)
- DBLinkState ∗ DBgetLinkState (DataBase ∗, long, long)
- int DBsetLinkState (DataBase ∗, long, long, DBLinkState ∗)
- LongList ∗ DBgetNodeInNeighb (DataBase ∗, long)
- LongList ∗ DBgetNodeOutNeighb (DataBase ∗, long)
- void DBprintDB (DataBase ∗)

4.13.1 Function Documentation

4.13.1.1 int DBaddLink (DataBase ∗, long, long, long, DBLinkState ∗)

Definition at line 1555 of file database-oli.c.

References addError(), ANDERROR, DBNodeVec_::cont, LongVec_::cont, CRITICAL, DBlinkDestroy(), DBlinkNew(), DBlinkStateCopy(), DBlinkTabSet(), DBnodeVecGet, DBLink_::id, DBNode_::inNeighb, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, longListPushBack, longListSort(), longVecSet(), max, DataBase_::nbLinks, DataBase_::nodeVec, DBNode_::outNeighb, LongVec_::size, DBLink_::state, and LongVec_::top.

```
{
    DBLink* link=NULL;
    int ret=0;

    if (dataBase == NULL || initLinkState==NULL
        || id <0 || src<0 || dst<0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (((id<dataBase->linkSrcVec.size) && (dataBase->linkSrcVec.cont[id]>0))
        ||
        ((id<dataBase->linkDstVec.size) && (dataBase->linkDstVec.cont[id]>0)))
    {
        addError(CRITICAL,"Trying to add a link with a reserved ID (ID=%ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return -1;
    }

    if ((link = DBlinkNew()) == NULL)
    {
        addError(CRITICAL,"Unable to create link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    link->id=id;

    if (DBlinkStateCopy(&(link->state), initLinkState))
    {
        addError(CRITICAL,"Unable to create link in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    if ((DBnodeVecGet(&(dataBase->nodeVec),src) == NULL) ||
        (DBnodeVecGet(&(dataBase->nodeVec),dst) == NULL))
    {
        addError(CRITICAL,"Source or destination node doesn’t exist in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    if (DBlinkTabSet(&(dataBase->linkTab),link,src,dst)<0)
    {
        addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %d
                 __FILE__,__LINE__);
        DBlinkDestroy(link);
        return -1;
    }

    ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
    ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[dst]->inNeighb),src));

    ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[src]->outNeighb)));
    ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[dst]->inNeighb)));

    ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,src+1));
    ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,dst+1));

    // Maximum non-null element
    dataBase->linkSrcVec.top = max(dataBase->linkSrcVec.top, id+1);
    dataBase->linkDstVec.top = dataBase->linkSrcVec.top;

    if (ret<0)
    {
        addError(CRITICAL,"Link addition uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks++;

    return ret;
}
```

4.13.1.2 int DBaddLSP (DataBase *, DBLabelSwitchedPath *, LongList *)

Definition at line 1679 of file database-oli.c.

References addError(), ANDERROR, chooseReroutedLSPs(), LongVec_::cont, CRITICAL, DBgetLinkLSPs(), DBlinkTabGet, DBlspCopy(), DBlspDestroy(), DBlspListInsert(), DBlspNew(), DBlspVecGet, DBlspVecSet(), DBupdateLSOnSetup(), FALSE, GLOBAL_BACK, DBLabelSwitchedPath_::id, DataBase_::id, DBLink_::id, isValidLSPLink(), DataBase_::linkSrcVec, DataBase_::linkTab, LOCAL_BACK, longListEnd, longListInit, longListMerge(), longVecEnd(), longVecInit(), DBLink_::lspList, DataBase_::lspVec, NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, DBLabelSwitchedPath_::precedence, DBLabelSwitchedPath_::primPath, LongVec_::size, DBLink_::state, LongVec_::top, TRUE, DBLabelSwitchedPath_::type, and WARNING.

```
{
    DBLabelSwitchedPath *newLSP, *contentLSP=NULL;
    DBLSPList *lspList;
    int i,ret=0;
    DBLink *lnk=NULL;
    LongVec isProcessed;
    double rerouteGain[NB_OA];
    bool allowLSP=TRUE;
#if defined SIMULATOR
    LongList idList;
#elif defined AGENT
    int j;
    bool inPath=FALSE;
#endif

#if defined LINUX && defined TIME2
    struct timezone tz;
    struct timeval  t1,t2;
#endif

    if (dataBase == NULL || lsp==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

#if defined LINUX && defined TIME2
    gettimeofday(&t1, &tz);
#endif

    if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
    {
        addError(CRITICAL,"Unable to initialize vector of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(rerouteGain,0,NB_OA*sizeof(double));

    // Check if establishment is possible
#if defined SIMULATOR
    if (longListInit(&(idList),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize list of longs in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    for (i=0;(i<lsp->path.top-1) && allowLSP;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
                                             &(lnk->state),lsp,rerouteGain);
        if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        idList.top=0;
        if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
        {
            addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        if (longListMerge(&(idList),preemptList,preemptList)<0)
        {
            addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&(idList));
            longVecEnd(&(isProcessed));
            return -1;
        }
        isProcessed.cont[lnk->id] = 1;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;(i<lsp->primPath.top-1) && allowLSP;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[i],lsp->primPath.co
                                                     &(lnk->state),lsp,rerouteGain);
                if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))=
                {
                    addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at l
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                idList.top=0;
                if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<
                {
                    addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                if (longListMerge(&(idList),preemptList,preemptList)<0)
                {
                    addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&(idList));
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
    longListEnd(&(idList));
#elif defined AGENT
    for (i=0;(i<lsp->path.top-1) && (lsp->path.cont[i]!=dataBase->id);i++);

    if (i<lsp->path.top-1)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
                                             &(lnk->state),lsp,rerouteGain);
        if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
                     __FILE__,__LINE__);
            longVecEnd(&(isProcessed));
        }
        if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)<0)
        {
            addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                     __FILE__,__LINE__);
            longVecEnd(&(isProcessed));
            return -1;
        }
        isProcessed.cont[lnk->id] = 1;
        inPath=TRUE;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (j=0;(j<lsp->primPath.top-1) && (lsp->primPath.cont[j]!=dataBase->id);j++);

        if (j<lsp->primPath.top-1)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[j],
                             lsp->primPath.cont[j+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[j],lsp->primPath.cor
                                                     &(lnk->state),lsp,rerouteGain);
                if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==
                {
                    addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at li
                             __FILE__,__LINE__);
                    longVecEnd(&(isProcessed));
                }
                if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)
                {
                    addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
                             __FILE__,__LINE__);
                    longVecEnd(&(isProcessed));
                    return -1;
                }
                isProcessed.cont[lnk->id] = 1;
            }
            inPath=TRUE;
        }
    }
    if (!inPath)
    {
        addError(CRITICAL,"Agent not concerned by this LSP in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    if (!allowLSP)
    {
        addError(CRITICAL,"LSP refused by the predicate in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }


    if ((newLSP=DBlspNew())==NULL)
    {
        addError(CRITICAL,"Unable to create LSP in %s at line %d",
                 __FILE__,__LINE__);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (DBlspCopy(newLSP,lsp)<0)
    {
        addError(CRITICAL,"Unable to create a valid LSP copy in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(newLSP);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (DBlspVecSet(&(dataBase->lspVec),newLSP,newLSP->id)<0)
    {
        addError(CRITICAL,"Unable to insert LSP in the general LSP container in %s at line %d",
                 __FILE__,__LINE__);
        DBlspDestroy(newLSP);
        longVecEnd(&(isProcessed));
        return -1;
    }

    if (newLSP->noContentionId>=0)
    {
        if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),newLSP->noContentionId))==NULL)
        {
            addError(WARNING,"Unable to get no contention LSP in %s at line %d",
                     __FILE__,__LINE__);
            newLSP->noContentionId=-1;
            // not critical enough to abort
        }
        else
        {
            contentLSP->noContentionId=newLSP->id;
        }
    }

    for (i=0;i<isProcessed.size;i++)
    {
        isProcessed.cont[i]=0;
    }


#if defined SIMULATOR
    // Add the LSP to each link list and update all the linkstates (only once !!!!!)
    for (i=0;i<newLSP->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
                         newLSP->path.cont[i+1]);
        ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
        ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->path.cont[i],
                                       newLSP->path.cont[i+1], &(lnk->state), newLSP));
        isProcessed.cont[lnk->id] = 1;
    }
    if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
    {
        for (i=0;i<newLSP->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
                             newLSP->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->primPath.cont[i],
                                               newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
#elif defined AGENT
    // Add the LSP to the link attached to the agent and update the linkstate
    for (i=0;i<newLSP->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
                         newLSP->path.cont[i+1]);
        ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));

        if (newLSP->path.cont[i] == dataBase->id)
        {
            ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->path.cont[i],
                                           newLSP->path.cont[i+1], &(lnk->state), newLSP));
            isProcessed.cont[lnk->id] = 1;
        }
    }
    if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
    {
        for (i=0;i<newLSP->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
                             newLSP->primPath.cont[i+1]);

            if (newLSP->primPath.cont[i] == dataBase->id)
            {
                if (isProcessed.cont[lnk->id] == 0)
                {
                    ANDERROR(ret,DBupdateLSOnSetup(dataBase, newLSP->primPath.cont[i],
                                                   newLSP->primPath.cont[i+1], &(lnk->state), newLSP)
                }
                break;
            }
        }
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif
```

```

    longVecEnd(&(isProcessed));

#if defined LINUX && defined TIME2
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time to add a new LSP : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif


    if (ret<0)
    {
        addError(CRITICAL,"LSP addition uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.13.1.3 int DBaddNode (DataBase ∗, long)

Definition at line 1466 of file database-oli.c.

References addError(), CRITICAL, DBnodeDestroy(), DBnodeNew(), DBnodeVecSet(), DBNode_::id, DataBase_::nbNodes, and DataBase_::nodeVec.

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((node=DBnodeNew()) == NULL)
    {
        addError(CRITICAL,"Unable to create node in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    node->id=id;

    if (DBnodeVecSet(&(dataBase->nodeVec),node,id) < 0)
    {
        addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %
                 __FILE__,__LINE__);
        DBnodeDestroy(node);
        return -1;
    }

    dataBase->nbNodes++;

    return 0;
}
```

4.13.1.4 int DBdestroy (DataBase ∗)

Definition at line 1349 of file database-oli.c.

References addError(), CRITICAL, DBlinkTabEnd(), DBlspVecEnd(), DBnodeVecEnd(), free, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, longVecEnd(), DataBase_::lspVec, and DataBase_::nodeVec.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBnodeVecEnd(&(dataBase->nodeVec));
    DBlspVecEnd(&(dataBase->lspVec));
    DBlinkTabEnd(&(dataBase->linkTab));
    longVecEnd(&(dataBase->linkSrcVec));
    longVecEnd(&(dataBase->linkDstVec));

    free(dataBase);

    return 0;
}
```

4.13.1.5 int DBevalLSOnRemove (DataBase ∗, long, long, DBLinkState ∗, DBLinkState ∗, LSPRequest ∗)

Definition at line 1259 of file database-oli.c.

References evalLS(), and REMOVE.

Referenced by computeBackup().

```
{
    return evalLS(dataBase, src, dst, newLS, oldLS, req, REMOVE);
}
```

4.13.1.6 int DBevalLSOnSetup (DataBase ∗, long, long, DBLinkState ∗, DBLinkState ∗, LSPRequest ∗)

Definition at line 1253 of file database-oli.c.

References evalLS(), and SETUP.

Referenced by computeBackup(), computeCost(), and isValidRequestLink().

```
{
    return evalLS(dataBase, src, dst, newLS, oldLS, req, SETUP);
}
```

4.13.1.7 long DBgetID (DataBase ∗)

Definition at line 1369 of file database-oli.c.

References addError(), CRITICAL, and DataBase_::id.

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->id;
}
```

4.13.1.8 long DBgetLinkDst (DataBase ∗, long)

Definition at line 1421 of file database-oli.c.

References addError(), CRITICAL, DataBase_::linkDstVec, and longVecGet().

Referenced by computeBackup().

```
{
    long ret;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecGet(&(dataBase->linkDstVec),id,&ret)<0)
    {
        addError(CRITICAL,"Inexistent link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return (ret-1);
}
```

4.13.1.9 long DBgetLinkID (DataBase ∗, long, long)

Definition at line 1381 of file database-oli.c.

References addError(), CRITICAL, DBlinkTabGet, DBLink_::id, and DataBase_::linkTab.

Referenced by computeBackup(), computeCost(), DBprintDB(), DBremoveLink(), and updateLS().

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL || src < 0 || dst < 0)
    {
        addError(CRITICAL,"Bad argument (NULL or negative value) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst))==NULL)
    {
        return -1;
    }

    return lnk->id;
}
```

4.13.1.10 DBLSPList∗ DBgetLinkLSPs (DataBase ∗, long, long)

Definition at line 2138 of file database-oli.c.

References addError(), CRITICAL, DBlinkTabGet, DataBase_::linkTab, and DBLink_::lspList.

Referenced by DBaddLSP().

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return NULL;
    }

    return &(lnk->lspList);
}
```

4.13.1.11 long DBgetLinkSrc (DataBase ∗, long)

Definition at line 1400 of file database-oli.c.

References addError(), CRITICAL, DataBase_::linkSrcVec, and longVecGet().

Referenced by computeBackup().

```
{
    long ret;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecGet(&(dataBase->linkSrcVec),id,&ret)<0)
    {
        addError(CRITICAL,"Inexistent link in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return (ret-1);
}
```

4.13.1.12 DBLinkState∗ DBgetLinkState (DataBase ∗, long, long)

Definition at line 2159 of file database-oli.c.

References addError(), CRITICAL, DBlinkTabGet, DataBase_::linkTab, and DBLink_::state.

Referenced by computeBackup(), and fillTopo().

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return NULL;
    }

    return &(lnk->state);
}
```

4.13.1.13 DBLabelSwitchedPath∗ DBgetLSP (DataBase ∗, long)

Definition at line 2125 of file database-oli.c.

References addError(), CRITICAL, DBlspVecGet, and DataBase_::lspVec.

Referenced by computeBackup(), evalLS(), and updateLS().

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    return DBlspVecGet(&(dataBase->lspVec), id);
}
```

4.13.1.14 long DBgetMaxNodeID (DataBase ∗)

Definition at line 1454 of file database-oli.c.

References addError(), CRITICAL, DataBase_::nodeVec, and DBNodeVec_::top.

Referenced by fillTopo().

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nodeVec.top-1;
}
```

4.13.1.15 long DBgetNbLinks (DataBase *)

Definition at line 1543 of file database-oli.c.

References addError(), CRITICAL, and DataBase_::nbLinks.

Referenced by fillTopo().

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nbLinks;
}
```

4.13.1.16 long DBgetNbNodes (DataBase *)

Definition at line 1442 of file database-oli.c.

References addError(), CRITICAL, and DataBase_::nbNodes.

Referenced by fillTopo().

```
{
    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return dataBase->nbNodes;
}
```

4.13.1.17 LongList* DBgetNodeInNeighb (DataBase *, long)

Definition at line 2209 of file database-oli.c.

References addError(), CRITICAL, DBnodeVecGet, DBNode_::inNeighb, and DataBase_::nodeVec.

Referenced by computeBackup(), and fillTopo().

```
{
    DBNode *node=NULL;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
    {
        addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return NULL;
    }

    return (&(node->inNeighb));
}
```

4.13.1.18 LongList∗ DBgetNodeOutNeighb (DataBase ∗, long)

Definition at line 2231 of file database-oli.c.

References addError(), CRITICAL, DBnodeVecGet, DataBase_::nodeVec, and DBNode_::outNeighb.

Referenced by computeBackup(), and fillTopo().

```
 {
     DBNode *node=NULL;

     if (dataBase == NULL)
     {
         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
         return NULL;
     }

     if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
     {
         addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
                  id,__FILE__,__LINE__);
         return NULL;
     }

     return (&(node->outNeighb));
 }
```

4.13.1.19 int DBlinkStateCopy (DBLinkState ∗, DBLinkState ∗)

Definition at line 660 of file database-oli.c.

References addError(), ANDERROR, DBLinkState_::bbw, DBLinkState_::cap, DBLinkState_::color, CRITICAL, dblVecCopy(), DBLinkState_::fbw, NB_OA, NB_PREEMPTION, DBLinkState_::pbw, DBLinkState_::rbw, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeBackup(), DBaddLink(), DBsetLinkState(), and evalLS().

```
 {
     int i,j,ret=0;

     if (dst == NULL || src == NULL)
     {
         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
         return -1;
     }
```

```

    dst->color=src->color;
    memcpy(&(dst->cap),&(src->cap),NB_OA * sizeof(double));
    memcpy(&(dst->rbw),&(src->rbw),NB_OA * NB_PREEMPTION * sizeof(double));
    memcpy(&(dst->pbw),&(src->pbw),NB_OA * NB_PREEMPTION * sizeof(double));

    for (i=0;(i<NB_OA && ret>=0);i++)
        for (j=0;(j<NB_PREEMPTION && ret>=0);j++)
        {
            ANDERROR(ret,dblVecCopy(&(dst->bbw[i][j]),&(src->bbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->remoteBbw[i][j]),&(src->remoteBbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->fbw[i][j]),&(src->fbw[i][j])));
            ANDERROR(ret,dblVecCopy(&(dst->remoteFbw[i][j]),&(src->remoteFbw[i][j])));
        }

    if (ret<0)
    {
        addError(CRITICAL,"Link state copy uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.13.1.20 int DBlinkStateDestroy (DBLinkState *)

Definition at line 613 of file database-oli.c.

References addError(), DBLinkState_::bbw, CRITICAL, dblVecEnd(), DBLinkState_::fbw, free, NB_OA, NB_PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeBackup().

```
{
    int i,j;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            dblVecEnd(&(ls->bbw[i][j]));
            dblVecEnd(&(ls->remoteBbw[i][j]));
            dblVecEnd(&(ls->fbw[i][j]));
            dblVecEnd(&(ls->remoteFbw[i][j]));
        }
    free(ls);

    return 0;
}
```

4.13.1.21 int DBlinkStateEnd (DBLinkState *)

Definition at line 637 of file database-oli.c.

References addError(), DBLinkState_::bbw, CRITICAL, dblVecEnd(), DBLinkState_::fbw, NB_OA, NB_-PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeCost(), DBlinkDestroy(), DBlinkEnd(), DBlinkInit(), DBlinkNew(), and isValid-RequestLink().

```
{
    int i,j;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            dblVecEnd(&(ls->bbw[i][j]));
            dblVecEnd(&(ls->remoteBbw[i][j]));
            dblVecEnd(&(ls->fbw[i][j]));
            dblVecEnd(&(ls->remoteFbw[i][j]));
        }

    return 0;
}
```

4.13.1.22 int DBlinkStateInit (DBLinkState ∗)

Definition at line 530 of file database-oli.c.

References addError(), DBLinkState_::bbw, CRITICAL, dblVecEnd(), dblVecInit(), DBLinkState_::fbw, NB_OA, NB_PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

Referenced by computeCost(), DBlinkInit(), DBlinkNew(), and isValidRequestLink().

```
{
    int i,j,k,l;

    if (ls == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(ls, 0, sizeof(DBLinkState));

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
            {
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
            else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                dblVecEnd(&(ls->fbw[i][j]));
                for (k=i;k>=0;k++)
                    for (l=j-1;l>=0;l++)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
        }

    return 0;
}
```

4.13.1.23 DBLinkState* DBlinkStateNew ()

Definition at line 444 of file database-oli.c.

References addError(), DBLinkState_::bbw, calloc, CRITICAL, dblVecEnd(), dblVecInit(), DBLinkState_-::fbw, free, NB_OA, NB_PREEMPTION, DBLinkState_::remoteBbw, and DBLinkState_::remoteFbw.

```
{
    DBLinkState* ls;
    int i,j,k,l;

    if ((ls=calloc(1,sizeof(DBLinkState)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    for (i=0;i<NB_OA;i++)
        for (j=0;j<NB_PREEMPTION;j++)
        {
            if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
            {
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                 __FILE__,__LINE__);
                return NULL;
            }
            else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return NULL;
            }
            else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return NULL;
            }
            else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
            {
                dblVecEnd(&(ls->bbw[i][j]));
                dblVecEnd(&(ls->remoteBbw[i][j]));
                dblVecEnd(&(ls->fbw[i][j]));
                for (k=i;k>=0;k--)
                    for (l=j-1;l>=0;l--)
                    {
                        dblVecEnd(&(ls->bbw[k][l]));
                        dblVecEnd(&(ls->remoteBbw[k][l]));
                        dblVecEnd(&(ls->fbw[k][l]));
                        dblVecEnd(&(ls->remoteFbw[k][l]));
                    }
                free(ls);
                addError(CRITICAL,"Unable to create link state in %s at line %d",
                         __FILE__,__LINE__);
                return NULL;
            }
        }

    return ls;
}
```

4.13.1.24 int DBlspCompare (const DBLabelSwitchedPath *, const DBLabelSwitchedPath *)

Definition at line 357 of file database-oli.c.

References DBLabelSwitchedPath_::bw, DBLabelSwitchedPath_::id, and DBLabelSwitchedPath_::precedence.

Referenced by DBlspListInsert(), and DBlspListRemove().

```
{
    if (LSPa->precedence > LSPb->precedence)
        return 1;
    else if (LSPa->precedence < LSPb->precedence)
        return -1;
    else if (LSPa->bw[0] > LSPb->bw[0])
        return 1;
    else if (LSPa->bw[0] < LSPb->bw[0])
        return -1;
    else
    {
        if (LSPa->id < LSPb->id)
            return 1;
        else if (LSPa->id > LSPb->id)
            return -1;
    }

    return 0;
}
```

4.13.1.25 int DBlspCopy (DBLabelSwitchedPath *, DBLabelSwitchedPath *)

Definition at line 157 of file database-oli.c.

References addError(), ANDERROR, DBLabelSwitchedPath_::backLSPIDs, DBLabelSwitchedPath_::bw, CRITICAL, DBLabelSwitchedPath_::forbidLinks, DBLabelSwitchedPath_::id, longListCopy, NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, DBLabelSwitchedPath_::precedence, DBLabelSwitchedPath_::primID, DBLabelSwitchedPath_::primPath, and DBLabelSwitchedPath_::type.

Referenced by DBaddLSP().

```
{
    int ret=0;

    if (dst == NULL || src==NULL)
    {
```

```
163         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
164                  __FILE__,__LINE__);
165         return -1;
166     }
167
168     dst->id=src->id;
169     dst->precedence=src->precedence;
170     memcpy(dst->bw,src->bw, NB_OA * sizeof(double));
171     dst->noContentionId = src->noContentionId;
172     ANDERROR(ret,longListCopy(&(dst->forbidLinks),&(src->forbidLinks)));
173     ANDERROR(ret,longListCopy(&(dst->path),&(src->path)));
174     dst->type=src->type;
175     dst->primID=src->primID;
176     ANDERROR(ret,longListCopy(&(dst->primPath),&(src->primPath)));
177     ANDERROR(ret,longListCopy(&(dst->backLSPIDs),&(src->backLSPIDs)));
178
179     if (ret<0)
180     {
181         addError(CRITICAL,"Label switched path copy uncomplete in %s at line %d",
182                  __FILE__,__LINE__);
183     }
184
185     return ret;
186 }
```

4.13.1.26 int DBlspDestroy (DBLabelSwitchedPath ∗)

Definition at line 122 of file database-oli.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, CRITICAL, DBLabelSwitchedPath_::forbidLinks, free, longListEnd, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

Referenced by DBaddLSP(), DBlspVecDestroy(), DBlspVecEnd(), DBlspVecResize(), and evalLS().

```
123 {
124     if (lsp == NULL)
125     {
126         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
127                  __FILE__,__LINE__);
128         return -1;
129     }
130
131     longListEnd(&(lsp->backLSPIDs));
132     longListEnd(&(lsp->primPath));
133     longListEnd(&(lsp->path));
134     longListEnd(&(lsp->forbidLinks));
135     free(lsp);
136
137     return 0;
138 }
```

4.13.1.27 int DBlspEnd (DBLabelSwitchedPath ∗)

Definition at line 140 of file database-oli.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, CRITICAL, DBLabelSwitchedPath_::forbidLinks, longListEnd, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

```
141 {
142     if (lsp == NULL)
143     {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListEnd(&(lsp->backLSPIDs));
    longListEnd(&(lsp->primPath));
    longListEnd(&(lsp->path));
    longListEnd(&(lsp->forbidLinks));

    return 0;
}
```

4.13.1.28 int DBlspInit (DBLabelSwitchedPath ∗)

Definition at line 73 of file database-oli.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, DBLabelSwitchedPath_::bw, CRITICAL, DBLabelSwitchedPath_::forbidLinks, longListEnd, longListInit, NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

```
{
    if (lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->forbidLinks),-1)<0)
    {
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->path),-1)<0)
    {
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->primPath),-1)<0)
    {
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(lsp->backLSPIDs),-1)<0)
    {
        longListEnd(&(lsp->primPath));
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(lsp->bw, 0, NB_OA * sizeof(double));
    lsp->noContentionId=-1;   //very important

    return 0;
}
```

4.13.1.29 int DBlspListDestroy (DBLSPList *)

Definition at line 251 of file database-oli.c.

References addError(), DBLSPList_::cont, CRITICAL, and free.

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }

    free(list->cont);
    free(list);

    return 0;
}
```

4.13.1.30 int DBlspListEnd (DBLSPList *)

Definition at line 266 of file database-oli.c.

References addError(), DBLSPList_::cont, CRITICAL, free, DBLSPList_::size, and DBLSPList_::top.

Referenced by DBlinkDestroy(), and DBlinkEnd().

```
{
    if (list == NULL || list->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }

    free(list->cont);
    list->cont = NULL;
    list->size = 0;
    list->top = 0;

    return 0;
}
```

4.13.1.31 int DBlspListInit (DBLSPList *, long)

Definition at line 223 of file database-oli.c.

References addError(), calloc, DBLSPList_::cont, CRITICAL, LSPLIST_INITSIZE, DBLSPList_::size, and DBLSPList_::top.

Referenced by DBlinkInit(), and DBlinkNew().

```
{
    void* ptr=NULL;

    if (list == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LSPLIST_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    list->size = size;
    list->top = 0;
    list->cont = ptr;

    return 0;
}
```

4.13.1.32 int DBlspListInsert (DBLSPList ∗, DBLabelSwitchedPath ∗)

Definition at line 283 of file database-oli.c.

References addError(), DBLSPList_::cont, CRITICAL, DBlspCompare(), realloc, DBLSPList_::size, and DBLSPList_::top.

Referenced by DBaddLSP().

```
{
    int a,b;
    void *ptr=NULL;

    if (list == NULL || list->cont == NULL || lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // check the capacity of the list
    if (list->top >= list->size)
    {
        if ((ptr = realloc(list->cont, list->size
                           * 2 * sizeof(DBLabelSwitchedPath*))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            list->cont=ptr;
            list->size*=2;
        }
    }

    // find the position in the list (to keep it sorted)
    a = 0;
    b = list->top-1;

    // empty list or after the last elem
    if (list->top == 0 || DBlspCompare(list->cont[b], lsp) >= 0)
    {
        list->cont[list->top++] = lsp;
        return (list->top-1);
    }

    // before the first elem
    if (DBlspCompare(lsp, list->cont[a]) >= 0)
    {
        memmove(list->cont+1, list->cont, (list->top)*sizeof(void*));
        list->cont[0] = lsp;
        list->top++;
        return 0;
    }

    // now the insert position is inside ]a,b[
    while (b - a > 1)
    {
        int mid = (a + b)/2;
        int ret = DBlspCompare(lsp, list->cont[mid]);

        if (ret == 1)
            b = mid;
        else if (ret == -1)
            a = mid;
        else // if (ret == 0)
        {
            a = mid;
            b = mid;
        }
    }

    // now insert before b
    memmove(list->cont+b+1, list->cont+b, (list->top - b)*sizeof(void*));
    list->cont[b] = lsp;
    list->top++;

    return b;
}
```

4.13.1.33 DBLSPList* DBlspListNew (long)

Definition at line 193 of file database-oli.c.

References addError(), calloc, DBLSPList_::cont, CRITICAL, free, LSPLIST_INITSIZE, DBLSPList_::size, and DBLSPList_::top.

```
{
    DBLSPList *list=NULL;
    void* ptr=NULL;

    if ((list = calloc(1,sizeof(DBLSPList))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
```

```
        size = LSPLIST_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(list);
        return NULL;
    }

    list->size = size;
    list->top = 0;
    list->cont = ptr;

    return list;
}
```

4.13.1.34 int DBlspListRemove (DBLSPList *, DBLabelSwitchedPath *)

Definition at line 378 of file database-oli.c.

References addError(), DBLSPList_::cont, CRITICAL, DBlspCompare(), DBLSPList_::top, and WARNING.

```
{
    int a,b,index;

    if (list == NULL || list->cont == NULL || lsp == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // find the position in the list
    a = 0;
    b = list->top-1;

    // empty list
    if (list->top == 0)
    {
        addError(WARNING,"Removing inexistent LSP in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    while (b - a > 1)
    {
        int mid = (a + b)/2;
        int ret = DBlspCompare(lsp, list->cont[mid]);

        if (ret == 1)
            b = mid;
        else if (ret == -1)
            a = mid;
        else // if (ret == 0)
        {
            a = mid;
            b = mid;
        }
    }

    if (DBlspCompare(lsp, list->cont[a]) == 0)
    {
        index = a;
    }
    else if (DBlspCompare(lsp, list->cont[b]) == 0)
    {
        index = b;
    }
    else // not found
    {
        addError(WARNING,"Removing inexistent LSP in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // now delete index
    memmove(list->cont + index, list->cont + index + 1, (list->top - index -1)*sizeof(void*));
    list->top--;

    return 0;
}
```

4.13.1.35 DBLabelSwitchedPath* DBlspNew ()

Definition at line 19 of file database-oli.c.

References addError(), DBLabelSwitchedPath_::backLSPIDs, calloc, CRITICAL, DBLabelSwitchedPath_::forbidLinks, free, longListEnd, longListInit, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, and DBLabelSwitchedPath_::primPath.

```
{
    DBLabelSwitchedPath* lsp;

    if ((lsp=calloc(1,sizeof(DBLabelSwitchedPath)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->forbidLinks),-1)<0)
    {
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->path),-1)<0)
    {
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->primPath),-1)<0)
    {
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(lsp->backLSPIDs),-1)<0)
    {
        longListEnd(&(lsp->primPath));
        longListEnd(&(lsp->path));
        longListEnd(&(lsp->forbidLinks));
        free(lsp);
        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    lsp->noContentionId=-1; //very important

    return lsp;
}
```

4.13.1.36 DataBase∗ DBnew (long)

Definition at line 1280 of file database-oli.c.

References addError(), calloc, CRITICAL, DBlinkTabEnd(), DBlinkTabInit(), DBlspVecEnd(), DBlspVecInit(), DBnodeVecEnd(), DBnodeVecInit(), free, DataBase_::id, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, LINKTAB_INITSIZE, longVecEnd(), longVecInit(), DataBase_::lspVec, DataBase_::nbLinks, DataBase_::nbNodes, and DataBase_::nodeVec.

```
{
    DataBase *dataBase=NULL;

    if ((dataBase=calloc(1,sizeof(DataBase)))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    dataBase->id=ID;

    if (DBnodeVecInit(&(dataBase->nodeVec),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general node container in %s at line %d",
                 __FILE__,__LINE__);
        free(dataBase);
        return NULL;
    }

    if (DBlspVecInit(&(dataBase->lspVec),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general LSP container in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        free(dataBase);
        return NULL;
    }

    if (DBlinkTabInit(&(dataBase->linkTab),-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the general link container in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        free(dataBase);
        return NULL;
    }

    if (longVecInit(&(dataBase->linkSrcVec),LINKTAB_INITSIZE)<0)
    {
        addError(CRITICAL,"Unable to initialize the link id-src translater in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        DBlinkTabEnd(&(dataBase->linkTab));
        free(dataBase);
        return NULL;
    }

    if (longVecInit(&(dataBase->linkDstVec),LINKTAB_INITSIZE)<0)
    {
        addError(CRITICAL,"Unable to initialize the link id-dst translater in %s at line %d",
                 __FILE__,__LINE__);
        DBnodeVecEnd(&(dataBase->nodeVec));
        DBlspVecEnd(&(dataBase->lspVec));
        DBlinkTabEnd(&(dataBase->linkTab));
        longVecEnd(&(dataBase->linkSrcVec));
        free(dataBase);
        return NULL;
    }

    dataBase->nbNodes=0;
    dataBase->nbLinks=0;

    return dataBase;
}
```

4.13.1.37 void DBprintDB (DataBase ∗)

Definition at line 2253 of file database-oli.c.

References DBLinkTab_::cont, DBNodeVec_::cont, DBgetLinkID(), DBprintLink(), DBprintNode(), DataBase_::linkTab, DataBase_::nodeVec, DBLinkTab_::size, and DBNodeVec_::size.

```
{
    long i,j;

    printf("Printing info about nodes ...\n");
    printf("----------------------------\n");

    for (i=0; i<db->nodeVec.size; i++)
    {
        if (db->nodeVec.cont[i])
        {
            printf("Node id : %ld\n", i);
            printf("------------\n");
            DBprintNode(db->nodeVec.cont[i]);
        }
    }

    printf("\nPrinting info about links ...\n");
    printf("----------------------------\n");

    for (i=0; i<db->linkTab.size; i++)
        for (j=0; j<db->linkTab.size; j++)
        {
            if (db->linkTab.cont[i][j])
            {
                printf("Link %ld-%ld (id = %ld)\n", i, j, DBgetLinkID(db, i, j));
                printf("----------------------\n");

                DBprintLink(db->linkTab.cont[i][j]);

            }
        }
}
```

4.13.1.38 int DBremoveLink (DataBase *, long, long)

Definition at line 1635 of file database-oli.c.

References addError(), ANDERROR, LongVec_::cont, DBNodeVec_::cont, CRITICAL, DBgetLinkID(), DBlinkTabGet, DBlinkTabRemove(), DBnodeVecGet, DBNode_::inNeighb, DataBase_::linkDstVec, DataBase_::linkSrcVec, DataBase_::linkTab, longListRemove(), longVecSet(), DataBase_::nbLinks, DataBase_::nodeVec, DBNode_::outNeighb, and LongVec_::top.

Referenced by DBremoveNode().

```
{
    int id,ret=0;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((DBnodeVecGet(&(dataBase->nodeVec),src)==NULL) ||
        (DBnodeVecGet(&(dataBase->nodeVec),dst)==NULL) ||
        (DBlinkTabGet(&(dataBase->linkTab),src,dst)==NULL))
    {
        addError(CRITICAL,"Link doesn’t exist or database unconsistancy in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
    ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[dst]->inNeighb),src));

    ANDERROR(ret,DBlinkTabRemove(&(dataBase->linkTab),src,dst));

    id=DBgetLinkID(dataBase,src,dst);
    ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,0));
    ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,0));

    while (dataBase->linkSrcVec.cont[dataBase->linkSrcVec.top-1] == 0)
        dataBase->linkSrcVec.top--;

    if (ret<0)
    {
        addError(CRITICAL,"Link removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks--;

    return ret;
}
```

4.13.1.39 int DBremoveLSP (DataBase *, long)

Definition at line 2005 of file database-oli.c.

References addError(), ANDERROR, and DBlinkTabGet.

```
{
    DBLabelSwitchedPath *lsp=NULL, *contentLSP=NULL;
    int i,ret=0;
    DBLink *lnk=NULL;
    LongVec isProcessed;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lsp = DBlspVecGet(&(dataBase->lspVec), id)) == NULL)
    {
        addError(CRITICAL,"Trying to remove inexistent LSP (id = %ld) in %s at line %d",
                 id,__FILE__,__LINE__);
        return -1;
    }

    if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
    {
        addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

#if defined SIMULATOR
    // Remove the LSP from each link list and update all the linkstates
    for (i=0;i<lsp->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
        ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->path.cont[i],
                                        lsp->path.cont[i+1], &(lnk->state), lsp));
        isProcessed.cont[lnk->id] = 1;
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;i<lsp->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                             lsp->primPath.cont[i+1]);
            if (isProcessed.cont[lnk->id] == 0)
            {
                ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->primPath.cont[i],
                                                lsp->primPath.cont[i+1], &(lnk->state), lsp));
                isProcessed.cont[lnk->id] = 1;
            }
        }
    }
#elif defined AGENT
    // Remove the LSP to the link attached to the agent and update the linkstate
    for (i=0;i<lsp->path.top-1;i++)
    {
        lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
                         lsp->path.cont[i+1]);
        ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));

        if (lsp->path.cont[i] == dataBase->id)
        {
            ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->path.cont[i],
                                            lsp->path.cont[i+1], &(lnk->state), lsp));
            isProcessed.cont[lnk->id] = 1;
        }
    }
    if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
    {
        for (i=0;i<lsp->primPath.top-1;i++)
        {
            lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
                                 lsp->primPath.cont[i+1]);

            if (lsp->primPath.cont[i] == dataBase->id)
            {
                if (isProcessed.cont[lnk->id] == 0)
                {
                    ANDERROR(ret,DBupdateLSOnRemove(dataBase, lsp->primPath.cont[i],
                                                   lsp->primPath.cont[i+1], &(lnk->state), lsp));
                }
                break;
            }
        }
    }
#else
    // Generate an error;
    COMPILE_ERROR;
#endif

    longVecEnd(&(isProcessed));

    // remove the lsp from the global list
    ANDERROR(ret,DBlspVecRemove(&(dataBase->lspVec), id));

    if (lsp->noContentionId>=0)
    {
        if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),lsp->noContentionId))==NULL)
        {
            addError(WARNING,"Unable to get no contention LSP in %s at line %d",
                     __FILE__,__LINE__);
            // not critical enough to abort
        }
        contentLSP->noContentionId=-1;
    }

    // free the lsp
    DBlspDestroy(lsp);

    if (ret<0)
    {
        addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    return ret;
}
```

4.13.1.40 int DBremoveNode (DataBase *, long)

Definition at line 1499 of file database-oli.c.

References addError(), ANDERROR, LongVec_::cont, CRITICAL, DBnodeVecGet, DBnodeVecRemove(), DBremoveLink(), DBNode_::inNeighb, DataBase_::nbLinks, DataBase_::nodeVec, DBNode_::outNeighb, and LongVec_::top.

```
{
    DBNode *node=NULL;
    int ret=0;

    if (dataBase == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((node=DBnodeVecGet(&(dataBase->nodeVec),id)) == NULL)
    {
        addError(CRITICAL,"Trying to remove an inexistent node in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    // remember that DBremoveLink will update the neighbour list
    while(node->inNeighb.top > 0)
    {
        ANDERROR(ret,DBremoveLink(dataBase,node->inNeighb.cont[node->inNeighb.top-1],id));
    }

    // remember that DBremoveLink will update the neighbour list
    while(node->outNeighb.top > 0)
    {
        ANDERROR(ret,DBremoveLink(dataBase,id,node->outNeighb.cont[node->outNeighb.top-1]));
    }

    ANDERROR(ret,DBnodeVecRemove(&(dataBase->nodeVec),id));

    if (ret<0)
    {
        addError(CRITICAL,"Node removal uncomplete in %s at line %d",
                 __FILE__,__LINE__);
    }

    dataBase->nbLinks--;

    return ret;
}
```

4.13.1.41 int DBsetLinkState (DataBase *, long, long, DBLinkState *)

Definition at line 2180 of file database-oli.c.

References addError(), CRITICAL, DBlinkStateCopy(), DBlinkTabGet, DataBase_::linkTab, and DBLink_::state.

```
{
    DBLink *lnk=NULL;

    if (dataBase == NULL || newLS == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
    {
        addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
                 src,dst,__FILE__,__LINE__);
        return -1;
    }

2198     if (DBlinkStateCopy(&(lnk->state), newLS)<0)
2199     {
2200         addError(CRITICAL,"Impossible to set linkstate on link (src = %ld, dst = %ld) in %s at line %
2201                  src,dst,__FILE__,__LINE__);
2202         return -1;
2203     }
2204
2205     return 0;
2206 }
```

4.13.1.42 int DBupdateLSOnRemove (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗)

Definition at line 1269 of file database-oli.c.

References REMOVE, and updateLS().

```
1270 {
1271     return updateLS(dataBase, src, dst, ls, lsp, REMOVE);
1272 }
```

4.13.1.43 int DBupdateLSOnSetup (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗)

Definition at line 1264 of file database-oli.c.

References SETUP, and updateLS().

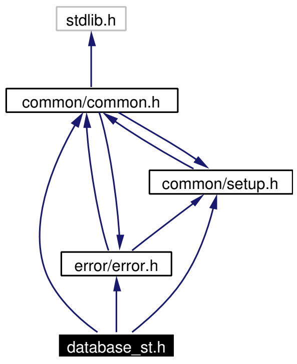
Referenced by DBaddLSP().

```
1265 {
1266     return updateLS(dataBase, src, dst, ls, lsp, SETUP);
1267 }
```

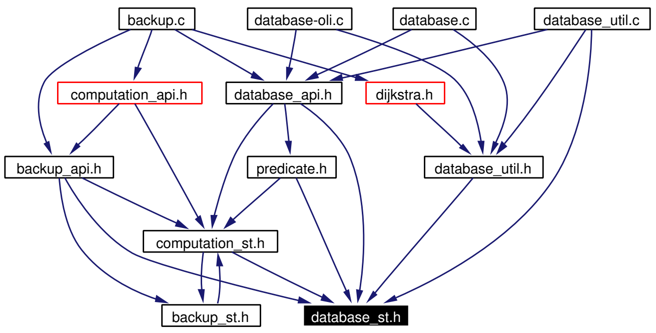
4.14 database_st.h File Reference

```
#include "common/common.h"
#include "common/setup.h"
#include "error/error.h"
```

Include dependency graph for database_st.h:



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



Data Structures

- struct DBLabelSwitchedPath_

  LSP structure.

- struct DBLinkState_

  Link state structure.

- struct DBLSPList_

Typedefs

- typedef DataBase_ DataBase
- typedef unsigned char DBLSPType

- typedef DBLabelSwitchedPath_ DBLabelSwitchedPath

  LSP structure.

- typedef DBLSPList_ DBLSPList
- typedef DBLinkState_ DBLinkState

  Link state structure.

Enumerations

- enum { PRIM, LOCAL_BACK, GLOBAL_BACK }

4.14.1 Typedef Documentation

4.14.1.1 typedef struct DataBase_ DataBase

Definition at line 8 of file database_st.h.

4.14.1.2 typedef struct DBLabelSwitchedPath_ DBLabelSwitchedPath

LSP structure.

Label Switched Path representation, used by DBaddLSP. It is often needed to translate LSPRequest (used when computing) to DBLabelSwitchedPath (used when adding a LSP to the database).

Referenced by DBlspVecResize().

4.14.1.3 typedef struct DBLinkState_ DBLinkState

Link state structure.

This is the information maintained for each link.

4.14.1.4 typedef struct DBLSPList_ DBLSPList

4.14.1.5 typedef unsigned char DBLSPType

Definition at line 18 of file database_st.h.

4.14.2 Enumeration Type Documentation

4.14.2.1 anonymous enum

Enumeration values:

PRIM

LOCAL_BACK

GLOBAL_BACK

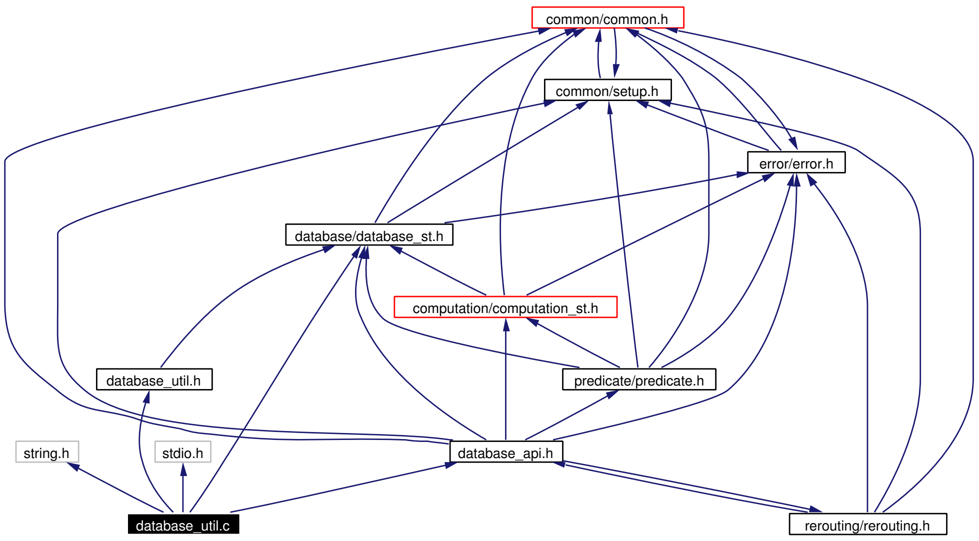
Definition at line 17 of file database_st.h.

```
17 {PRIM,LOCAL_BACK,GLOBAL_BACK};
```

4.15 database_util.c File Reference

```
#include "database_util.h"
#include "database_st.h"
#include "database_api.h"
#include <stdio.h>
#include <string.h>
```

Include dependency graph for database_util.c:



Functions

- DBNode ∗ DBnodeNew ()

  return a newly (dynamically) allocated DBNode object.

- int DBnodeInit (DBNode ∗node)

  initialize a DBNode object allready allocated somewhere else.

- int DBnodeDestroy (DBNode ∗node)

  clear function to free the ressources of a Link object allocated on the heap.

- int DBnodeEnd (DBNode ∗node)

  clear function to free the ressources of a Node object allocated on the stack.

- void DBprintNode (DBNode ∗node)
- DBLink ∗ DBlinkNew ()

  return a newly (dynamically) allocated Link object.

- int DBlinkInit (DBLink ∗link)

  initialize a Link object allready allocated somewhere else.

- int DBlinkDestroy (DBLink ∗link)

  clear function to free the ressources of a Link object allocated on the heap.

- int DBlinkEnd (DBLink ∗link)

  clear function to free the ressources of a Link object allocated on the stack.

- void DBprintLink (DBLink ∗link)
- DBNodeVec ∗ DBnodeVecNew (long size)
- int DBnodeVecInit (DBNodeVec ∗vec, long size)
- int DBnodeVecDestroy (DBNodeVec ∗vec)
- int DBnodeVecEnd (DBNodeVec ∗vec)
- int DBnodeVecResize (DBNodeVec ∗vec, long size)
- int DBnodeVecSet (DBNodeVec ∗vec, DBNode ∗node, long id)
- int DBnodeVecRemove (DBNodeVec ∗vec, long id)
- DBLSPVec ∗ DBlspVecNew (long size)
- int DBlspVecInit (DBLSPVec ∗vec, long size)
- int DBlspVecDestroy (DBLSPVec ∗vec)
- int DBlspVecEnd (DBLSPVec ∗vec)
- int DBlspVecResize (DBLSPVec ∗vec, long size)
- int DBlspVecSet (DBLSPVec ∗vec, DBLabelSwitchedPath ∗lsp, long id)
- int DBlspVecRemove (DBLSPVec ∗vec, long id)
- DBLinkTab ∗ DBlinkTabNew (long size)
- int DBlinkTabInit (DBLinkTab ∗tab, long size)
- int DBlinkTabDestroy (DBLinkTab ∗tab)
- int DBlinkTabEnd (DBLinkTab ∗tab)
- int DBlinkTabResize (DBLinkTab ∗tab, long size)
- int DBlinkTabSet (DBLinkTab ∗tab, DBLink ∗lnk, long src, long dst)
- int DBlinkTabRemove (DBLinkTab ∗tab, long src, long dst)

4.15.1 Function Documentation

4.15.1.1 int DBlinkDestroy (DBLink ∗ *link*)

clear function to free the ressources of a Link object allocated on the heap.

Definition at line 204 of file database_util.c.

References addError(), CRITICAL, DBlinkStateEnd(), DBlspListEnd(), free, DBLink_::lspList, and DBLink_::state.

Referenced by DBaddLink(), DBlinkTabDestroy(), DBlinkTabEnd(), DBlinkTabRemove(), and DBlink-TabResize().

```
205 {
206     if (link == NULL)
207     {
208         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
209                  __FILE__,__LINE__);
210         return -1;
211     }
212
213     DBlinkStateEnd(&link->state);
214     DBlspListEnd(&(link->lspList));
```

```
    free(link);

    return 0;
}
```

4.15.1.2 int DBlinkEnd (DBLink ∗ *link*)

clear function to free the ressources of a Link object allocated on the stack.

Definition at line 221 of file database_util.c.

References addError(), CRITICAL, DBlinkStateEnd(), DBlspListEnd(), DBLink_::lspList, and DBLink_::state.

```
{
    if (link == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBlinkStateEnd(&link->state);
    DBlspListEnd(&(link->lspList));

    return 0;
}
```

4.15.1.3 int DBlinkInit (DBLink ∗ *link*)

initialize a Link object allready allocated somewhere else.

Definition at line 176 of file database_util.c.

References addError(), CRITICAL, DBlinkStateEnd(), DBlinkStateInit(), DBlspListInit(), DBLink_::lspList, and DBLink_::state.

```
{
    if (link == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (DBlinkStateInit(&(link->state)) == -1)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (DBlspListInit(&(link->lspList),-1) < 0)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkStateEnd(&(link->state));
        return -1;
    }

    return 0;
}
```

4.15.1.4 DBLink* DBlinkNew ()

return a newly (dynamically) allocated Link object.

Definition at line 144 of file database_util.c.

References addError(), calloc, CRITICAL, DBlinkStateEnd(), DBlinkStateInit(), DBlspListInit(), free, DBLink_::lspList, and DBLink_::state.

Referenced by DBaddLink().

```
{
    DBLink* ptr=NULL;

    if ((ptr = calloc(1,sizeof(DBLink))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (DBlinkStateInit(&(ptr->state)) == -1)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        free(ptr);
        return NULL;
    }

    if (DBlspListInit(&(ptr->lspList),-1) < 0)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkStateEnd(&(ptr->state));
        free(ptr);
        return NULL;
    }

    return ptr;
}
```

4.15.1.5 int DBlinkTabDestroy (DBLinkTab * *tab*)

Definition at line 799 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), free, and DBLinkTab_::size.

```
{
    int i,j;

    if (tab == NULL || tab->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<tab->size;i++)
    {
        for (j=0; j<tab->size; j++)
        {
            if (tab->cont[i][j] != NULL)
            {
                DBlinkDestroy(tab->cont[i][j]);
            }
        }
        free(tab->cont[i]);
    }
    free(tab->cont);
    free(tab);

    return 0;
}
```

4.15.1.6 int DBlinkTabEnd (DBLinkTab ∗ *tab*)

Definition at line 827 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), free, and DBLinkTab_::size.

Referenced by DBdestroy(), and DBnew().

```
{
    int i,j;

    if (tab == NULL || tab->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<tab->size;i++)
    {
        for (j=0; j<tab->size; j++)
        {
            if (tab->cont[i][j] != NULL)
            {
                DBlinkDestroy(tab->cont[i][j]);
            }
        }
        free(tab->cont[i]);
    }

    free(tab->cont);
    tab->cont = NULL;
    tab->size = 0;

    return 0;
}
```

4.15.1.7 int DBlinkTabInit (DBLinkTab ∗ *tab*, long *size*)

Definition at line 756 of file database_util.c.

References addError(), calloc, DBLinkTab_::cont, CRITICAL, free, LINKTAB_INITSIZE, and DBLink-Tab_::size.

Referenced by DBnew().

```
{
    DBLink ***ptr=NULL;
    int i;

    if (tab == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LINKTAB_INITSIZE;

    if ((ptr=(calloc(size,sizeof(DBLink**))))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    else
    {
        for (i=0;i<size;i++)
        {
            if ((ptr[i]=(calloc(size, sizeof(DBLink*))))==NULL)
            {
                addError(CRITICAL,"Critical lack of memory in %s at line %d",
                         __FILE__,__LINE__);
                for (i=i-1;i>=0;i--)
                    free(ptr[i]);
                free(ptr);
                return -1;
            }
        }
    }

    tab->size=size;
    tab->cont=ptr;

    return 0;
}
```

4.15.1.8 DBLinkTab∗ DBlinkTabNew (long *size*)

Definition at line 710 of file database_util.c.

References addError(), calloc, DBLinkTab_::cont, CRITICAL, free, LINKTAB_INITSIZE, and DBLinkTab_::size.

```
{
    DBLinkTab *tab=NULL;
    DBLink ***ptr=NULL;
    int i;

    if ((tab = calloc(1,sizeof(DBLinkTab))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LINKTAB_INITSIZE;

    if ((ptr=(calloc(size,sizeof(DBLink**))))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(tab);
        return NULL;
    }
    else
    {
        for (i=0;i<size;i++)
        {
            if ((ptr[i]=(calloc(size, sizeof(DBLink*))))==NULL)
            {
                addError(CRITICAL,"Critical lack of memory in %s at line %d",
                         __FILE__,__LINE__);
                for (i=i-1;i>=0;i--)
                    free(ptr[i]);
                free(ptr);
                free(tab);
                return NULL;
            }
        }
    }

    tab->size=size;
    tab->cont=ptr;

    return tab;
}
```

4.15.1.9 int DBlinkTabRemove (DBLinkTab ∗ *tab*, long *src*, long *dst*)

Definition at line 963 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), and DBLinkTab_::size.

Referenced by DBremoveLink().

```
{
    if (tab == NULL || tab->cont == NULL ||
        src <0 || dst<0 || src >= tab->size || dst >= tab->size ||
        tab->cont[src][dst] == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBlinkDestroy(tab->cont[src][dst]);
    tab->cont[src][dst]=NULL;

    return 0;
}
```

4.15.1.10 int DBlinkTabResize (DBLinkTab ∗ *tab*, long *size*)

Definition at line 858 of file database_util.c.

References addError(), calloc, DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), free, min, realloc, and DBLinkTab_::size.

Referenced by DBlinkTabSet().

```
{
    DBLink*** ptr=NULL;
    DBLink** ptr2=NULL;
    int i,j;

    if (tab == NULL || tab->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size < tab->size)
    {
        for (i=size;i<tab->size;i++)
        {
            for (j=size; j<tab->size; j++)
            {
                if (tab->cont[i][j] != NULL)
                {
                    DBlinkDestroy(tab->cont[i][j]);
                    tab->cont[i][j]=NULL;
                }
            }
        }
        free(tab->cont[i]);
    }

    if ((ptr = realloc(tab->cont,size * sizeof(DBLink**))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    else
    {
        tab->cont = ptr;

        for (i=0;i<min(tab->size,size);i++)
        {
            if ((ptr2 = realloc(ptr[i], size * sizeof(DBLink*)))==NULL)
            {
                addError(CRITICAL,"Critical lack of memory in %s at line %d",
                         __FILE__,__LINE__);
                tab->size=min(tab->size,size);
                return -1;
            }

            ptr[i] = ptr2;

            if (size > tab->size)
            {
                memset(ptr2 + tab->size, 0, (size-tab->size) * sizeof(DBLink*));
            }
        }

        if (size > tab->size)
        {
            for (i=tab->size;i<size;i++)
            {
                if ((ptr[i] = calloc(size, sizeof(DBLink*)))==NULL)
                {
                    addError(CRITICAL,"Critical lack of memory in %s at line %d",
                             __FILE__,__LINE__);
                    tab->size=i;
                    return -1;
                }
            }
        }
    }

    tab->size=size;

    return 0;
}
```

4.15.1.11 int DBlinkTabSet (DBLinkTab * *tab*, DBLink * *lnk*, long *src*, long *dst*)

Definition at line 935 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkTabResize(), max, and DBLinkTab_::size.

Referenced by DBaddLink().

```
{
    long resize;

    if (tab == NULL || tab->cont == NULL || src <0 || dst<0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    resize=max(src,dst)+1;

    if (resize > tab->size)
    {
        if (DBlinkTabResize(tab,max(2*tab->size,resize))<0)
        {
            addError(CRITICAL,"Unable to resize link table prior to insertion in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    tab->cont[src][dst]=lnk;

    return 0;
}
```

4.15.1.12 int DBlspVecDestroy (DBLSPVec * *vec*)

Definition at line 565 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBlspDestroy(), free, and DBLSPVec_::size.

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBlspDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.15.1.13 int DBlspVecEnd (DBLSPVec ∗ *vec*)

Definition at line 590 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBlspDestroy(), free, and DBLSPVec_::size.

Referenced by DBdestroy(), and DBnew().

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBlspDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;

    return 0;
}
```

4.15.1.14 int DBlspVecInit (DBLSPVec ∗ *vec*, long *size*)

Definition at line 538 of file database_util.c.

References addError(), calloc, DBLSPVec_::cont, CRITICAL, LSPVEC_INITSIZE, and DBLSPVec_-::size.

Referenced by DBnew().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LSPVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->cont = ptr;

    return 0;
}
```

4.15.1.15 DBLSPVec∗ DBlspVecNew (long *size*)

Definition at line 509 of file database_util.c.

References addError(), calloc, DBLSPVec_::cont, CRITICAL, free, LSPVEC_INITSIZE, and DBLSPVec_::size.

```
{
    DBLSPVec *vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(DBLSPVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LSPVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->cont = ptr;

    return vec;
}
```

4.15.1.16 int DBlspVecRemove (DBLSPVec ∗ *vec*, long *id*)

Definition at line 689 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, and DBLSPVec_::size.

```
{
    if (vec == NULL || vec->cont == NULL ||
        id <0 || id >= vec->size || vec->cont[id] == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[id]=NULL;

    return 0;
}
```

4.15.1.17 int DBlspVecResize (DBLSPVec ∗ *vec*, long *size*)

Definition at line 616 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBLabelSwitchedPath, DBlspDestroy(), realloc, and DBLSPVec_::size.

Referenced by DBlspVecSet().

```
{
    void *ptr=NULL;
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size < vec->size)
    {
        for (i=size;i<vec->size;i++)
        {
            if (vec->cont[i]!=NULL)
            {
                DBlspDestroy(vec->cont[i]);
                vec->cont[i]=NULL;
            }
        }
    }

    if ((ptr = realloc(vec->cont, size * sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size > vec->size)
    {
        memset(ptr + (vec->size * sizeof(DBLabelSwitchedPath*)), 0, (size-vec->size) * sizeof(DBLabelS
    }

    vec->size=size;
    vec->cont=ptr;

    return 0;
}
```

4.15.1.18 int DBlspVecSet (DBLSPVec ∗ *vec*, DBLabelSwitchedPath ∗ *lsp*, long *id*)

Definition at line 658 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBlspVecResize(), max, and DBLSPVec_::size.

Referenced by DBaddLSP().

```
{
    if (vec == NULL || vec->cont == NULL || id <0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (id >= vec->size)
    {
        if (DBlspVecResize(vec,max(2*vec->size,id+1))<0)
        {
            addError(CRITICAL,"Unable to resize LSP vector prior to insertion in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (vec->cont[id] != NULL)
    {
        addError(CRITICAL,"Trying to add an LSP with a reserved ID in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[id]=lsp;

    return 0;
}
```

4.15.1.19 int DBnodeDestroy (DBNode ∗ *node*)

clear function to free the ressources of a Link object allocated on the heap.

Definition at line 77 of file database_util.c.

References addError(), CRITICAL, free, DBNode_::inNeighb, longListEnd, and DBNode_::outNeighb.

Referenced by DBaddNode(), DBnodeVecDestroy(), DBnodeVecEnd(), DBnodeVecRemove(), and DBnodeVecResize().

```
{
    if (node == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListEnd(&(node->inNeighb));
    longListEnd(&(node->outNeighb));
    free(node);

    return 0;
}
```

4.15.1.20 int DBnodeEnd (DBNode ∗ *node*)

clear function to free the ressources of a Node object allocated on the stack.

Definition at line 94 of file database_util.c.

References addError(), CRITICAL, DBNode_::inNeighb, longListEnd, and DBNode_::outNeighb.

```
{
    if (node == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListEnd(&(node->inNeighb));
    longListEnd(&(node->outNeighb));

    return 0;
}
```

4.15.1.21 int DBnodeInit (DBNode ∗ *node*)

initialize a DBNode object allready allocated somewhere else.

Definition at line 49 of file database_util.c.

References addError(), CRITICAL, DBNode_::inNeighb, longListEnd, longListInit, and DBNode_::outNeighb.

```
{
    if (node == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(node->inNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the incoming neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(node->outNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the outgoing neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&(node->inNeighb));
        return -1;
    }

    return 0;
}
```

4.15.1.22 DBNode∗ DBnodeNew ()

return a newly (dynamically) allocated DBNode object.

Definition at line 18 of file database_util.c.

References addError(), calloc, CRITICAL, free, DBNode_::inNeighb, longListEnd, longListInit, and DBNode_::outNeighb.

Referenced by DBaddNode().

```
{
    DBNode* ptr=NULL;

    if ((ptr = calloc(1,sizeof(DBNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(ptr->inNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the incoming neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(ptr->outNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the outgoing neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&(ptr->inNeighb));
        free(ptr);
        return NULL;
    }

    return ptr;
}
```

4.15.1.23 int DBnodeVecDestroy (DBNodeVec ∗ *vec*)

Definition at line 349 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeDestroy(), free, and DBNodeVec_::size.

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBnodeDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.15.1.24 int DBnodeVecEnd (DBNodeVec ∗ *vec*)

Definition at line 374 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeDestroy(), free, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBdestroy(), and DBnew().

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBnodeDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.15.1.25 int DBnodeVecInit (DBNodeVec ∗ *vec*, long *size*)

Definition at line 321 of file database_util.c.

References addError(), calloc, DBNodeVec_::cont, CRITICAL, NODEVEC_INITSIZE, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBnew().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = NODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBNode*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.15.1.26 DBNodeVec* DBnodeVecNew (long *size*)

Definition at line 290 of file database_util.c.

References addError(), calloc, DBNodeVec_::cont, CRITICAL, free, NODEVEC_INITSIZE, DBNodeVec_::size, and DBNodeVec_::top.

```
{
    DBNodeVec *vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(DBNodeVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = NODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBNode*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.15.1.27 int DBnodeVecRemove (DBNodeVec * *vec*, long *id*)

Definition at line 485 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeDestroy(), DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBremoveNode().

```
{
    if (vec == NULL || vec->cont == NULL ||
        id <0 || id >= vec->size || vec->cont[id] == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBnodeDestroy(vec->cont[id]);
    vec->cont[id]=NULL;

    while (vec->cont[vec->top-1] == NULL)
        vec->top--;

    return 0;
}
```

4.15.1.28 int DBnodeVecResize (DBNodeVec ∗ *vec*, long *size*)

Definition at line 401 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBNode, DBnodeDestroy(), realloc, DBNode-Vec_::size, and DBNodeVec_::top.

Referenced by DBnodeVecSet().

```
{
    void *ptr=NULL;
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size < vec->size)
    {
        for (i=size;i<vec->size;i++)
        {
            if (vec->cont[i]!=NULL)
            {
                DBnodeDestroy(vec->cont[i]);
                vec->cont[i]=NULL;
            }
        }

        if (size < vec->top)
        {
            vec->top = size;
            while (vec->cont[vec->top-1] == NULL)
                vec->top--;
        }

    }



    if ((ptr = realloc(vec->cont, size * sizeof(DBNode*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size > vec->size)
    {
        memset(ptr + (vec->size * sizeof(DBNode*)), 0, (size-vec->size) * sizeof(DBNode*));
    }

    vec->size=size;
    vec->cont=ptr;

    return 0;
}
```

4.15.1.29 int DBnodeVecSet (DBNodeVec ∗ *vec*, DBNode ∗ *node*, long *id*)

Definition at line 453 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeVecResize(), max, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBaddNode().

```
{
    if (vec == NULL || vec->cont == NULL || node == NULL || id <0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (id >= vec->size)
    {
        if (DBnodeVecResize(vec,max(2*vec->size,id+1))<0)
        {
            addError(CRITICAL,"Unable to resize node vector prior to insertion in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (vec->cont[id] != NULL)
    {
        addError(CRITICAL,"Trying to add a node with a reserved ID in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[id]=node;
    vec->top = max(vec->top, id+1);

    return 0;
}
```

4.15.1.30 void DBprintLink (DBLink ∗ *link*)

Definition at line 237 of file database_util.c.

References addError(), DBLinkState_::cap, DBLSPList_::cont, CRITICAL, DBLabelSwitchedPath_::id, DBLink_::lspList, NB_OA, NB_PREEMPTION, DBLinkState_::pbw, DBLinkState_::rbw, DBLink_::state, and DBLSPList_::top.

Referenced by DBprintDB().

```
{
    long i,oa;
    double ptot,rtot;
    DBLabelSwitchedPath* lsp=NULL;

    if (link == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return;
    }

    printf("\tList of LSPs\n");
    printf("\t------------\n");

    for (i=0; i<link->lspList.top; ++i)
    {
        lsp = link->lspList.cont[i];
        printf("%ld ", lsp->id);
    }

    printf("\n\n");

    printf("\tLink-state\n");
    printf("\t----------\n");

    for (oa=0; oa<NB_OA; ++oa)
    {
        ptot = 0;
        rtot = 0;

        printf("\tCapacity[%ld] = %f\n", oa, link->state.cap[oa]);

        for (i=0; i<NB_PREEMPTION; ++i)
        {
            ptot += link->state.pbw[oa][i];
            rtot += link->state.rbw[oa][i];
        }

        printf("\tpbw[%ld] = %f\n", oa, ptot);
        printf("\trbw[%ld] = %f\n", oa, rtot);
    }

    printf("\n\n");

}
```

4.15.1.31 void DBprintNode (DBNode ∗ *node*)

Definition at line 110 of file database_util.c.

References addError(), LongVec_::cont, CRITICAL, DBNode_::inNeighb, DBNode_::outNeighb, and LongVec_::top.

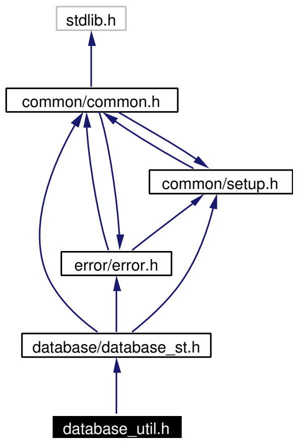
Referenced by DBprintDB().

```
{
    long i;

    if (node == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return;
    }

    printf("Incoming neighboors : \n");

    for (i=0; i<node->inNeighb.top; i++)
    {
        printf("%ld ", node->inNeighb.cont[i]);
    }

    printf("\nOutgoing neighboors : \n");

    for (i=0; i<node->outNeighb.top; i++)
    {
        printf("%ld ", node->outNeighb.cont[i]);
    }

    printf("\n");
}
```

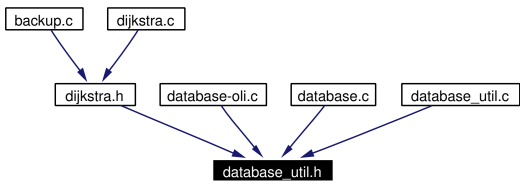
4.16 database_util.h File Reference

`#include "database/database_st.h"`

Include dependency graph for database_util.h:



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



Data Structures

- struct DataBase_
- struct DBLink_
- struct DBLinkTab_
- struct DBLSPVec_
- struct DBNode_
- struct DBNodeVec_

Defines

- #define DBnodeVecGet(a, b) (((b)>=((a) → size))?NULL:(((DBNodeVec∗)(a)) → cont[b]))
- #define DBlspVecGet(a, b) (((b)>=((a) → size))?NULL:(((DBLSPVec∗)(a)) → cont[b]))
- #define DBlinkTabGet(a, b, c) ((((b)>=(a) → size)||((c)>=(a) → size))?NULL:((a) → cont[b][c]))

Typedefs

- typedef DBNode_ DBNode

- typedef DBLink_ DBLink
- typedef DBNodeVec_ DBNodeVec
- typedef DBLSPVec_ DBLSPVec
- typedef DBLinkTab_ DBLinkTab

Functions

- DBNode ∗ DBnodeNew ()

  return a newly (dynamically) allocated DBNode object.

- int DBnodeEnd ()
- int DBnodeInit (DBNode ∗)

  initialize a DBNode object allready allocated somewhere else.

- int DBnodeDestroy (DBNode ∗)

  clear function to free the ressources of a Link object allocated on the heap.

- void DBprintNode (DBNode ∗)
- DBLink ∗ DBlinkNew ()

  return a newly (dynamically) allocated Link object.

- int DBlinkEnd ()
- int DBlinkInit (DBLink ∗)

  initialize a Link object allready allocated somewhere else.

- int DBlinkDestroy (DBLink ∗)

  clear function to free the ressources of a Link object allocated on the heap.

- void DBprintLink (DBLink ∗)
- DBNodeVec ∗ DBnodeVecNew (long)
- int DBnodeVecInit (DBNodeVec ∗, long)
- int DBnodeVecDestroy (DBNodeVec ∗)
- int DBnodeVecEnd (DBNodeVec ∗)
- int DBnodeVecResize (DBNodeVec ∗, long)
- int DBnodeVecSet (DBNodeVec ∗, DBNode ∗, long)
- int DBnodeVecRemove (DBNodeVec ∗, long)
- DBLSPVec ∗ DBlspVecNew (long)
- int DBlspVecInit (DBLSPVec ∗, long)
- int DBlspVecDestroy (DBLSPVec ∗)
- int DBlspVecEnd (DBLSPVec ∗)
- int DBlspVecResize (DBLSPVec ∗, long)
- int DBlspVecSet (DBLSPVec ∗, DBLabelSwitchedPath ∗, long)
- int DBlspVecRemove (DBLSPVec ∗, long)
- DBLinkTab ∗ DBlinkTabNew (long)
- int DBlinkTabInit (DBLinkTab ∗, long)
- int DBlinkTabDestroy (DBLinkTab ∗)
- int DBlinkTabEnd (DBLinkTab ∗)
- int DBlinkTabResize (DBLinkTab ∗, long)
- int DBlinkTabSet (DBLinkTab ∗, DBLink ∗, long, long)
- int DBlinkTabRemove (DBLinkTab ∗, long, long)

4.16.1 Define Documentation

4.16.1.1 #define DBlinkTabGet(a, b, c) ((((b)>=(a) → size)||((c)>=(a) → size))?NULL:((a) → cont[b][c]))

Definition at line 106 of file database_util.h.

Referenced by DBaddLSP(), DBgetLinkID(), DBgetLinkLSPs(), DBgetLinkState(), DBremoveLink(), DBremoveLSP(), and DBsetLinkState().

4.16.1.2 #define DBlspVecGet(a, b) (((b)>=((a) → size))?NULL:(((DBLSPVec∗)(a)) → cont[b]))

Definition at line 86 of file database_util.h.

Referenced by DBaddLSP(), and DBgetLSP().

4.16.1.3 #define DBnodeVecGet(a, b) (((b)>=((a) → size))?NULL:(((DBNodeVec∗)(a)) → cont[b]))

Definition at line 67 of file database_util.h.

Referenced by DBaddLink(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), DBremoveLink(), and DBremoveNode().

4.16.2 Typedef Documentation

4.16.2.1 typedef struct DBLink_ DBLink

4.16.2.2 typedef struct DBLinkTab_ DBLinkTab

4.16.2.3 typedef struct DBLSPVec_ DBLSPVec

4.16.2.4 typedef struct DBNode_ DBNode

Referenced by DBnodeVecResize().

4.16.2.5 typedef struct DBNodeVec_ DBNodeVec

4.16.3 Function Documentation

4.16.3.1 int DBlinkDestroy (DBLink ∗)

clear function to free the ressources of a Link object allocated on the heap.

Definition at line 204 of file database_util.c.

References addError(), CRITICAL, DBlinkStateEnd(), DBlspListEnd(), free, DBLink_::lspList, and DBLink_::state.

Referenced by DBaddLink(), DBlinkTabDestroy(), DBlinkTabEnd(), DBlinkTabRemove(), and DBlinkTabResize().

```
205 {
206     if (link == NULL)
207     {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBlinkStateEnd(&link->state);
    DBlspListEnd(&(link->lspList));
    free(link);

    return 0;
}
```

4.16.3.2 int DBlinkEnd ()

4.16.3.3 int DBlinkInit (DBLink ∗)

initialize a Link object allready allocated somewhere else.

Definition at line 176 of file database_util.c.

References addError(), CRITICAL, DBlinkStateEnd(), DBlinkStateInit(), DBlspListInit(), DBLink_::lspList, and DBLink_::state.

```
{
    if (link == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (DBlinkStateInit(&(link->state)) == -1)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (DBlspListInit(&(link->lspList),-1) < 0)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkStateEnd(&(link->state));
        return -1;
    }

    return 0;
}
```

4.16.3.4 DBLink∗ DBlinkNew ()

return a newly (dynamically) allocated Link object.

Definition at line 144 of file database_util.c.

References addError(), calloc, CRITICAL, DBlinkStateEnd(), DBlinkStateInit(), DBlspListInit(), free, DBLink_::lspList, and DBLink_::state.

Referenced by DBaddLink().

```
{
    DBLink* ptr=NULL;

    if ((ptr = calloc(1,sizeof(DBLink))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (DBlinkStateInit(&(ptr->state)) == -1)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        free(ptr);
        return NULL;
    }

    if (DBlspListInit(&(ptr->lspList),-1) < 0)
    {
        addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
                 __FILE__,__LINE__);
        DBlinkStateEnd(&(ptr->state));
        free(ptr);
        return NULL;
    }

    return ptr;
}
```

4.16.3.5 int DBlinkTabDestroy (DBLinkTab *)

Definition at line 799 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), free, and DBLinkTab_::size.

```
{
    int i,j;

    if (tab == NULL || tab->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<tab->size;i++)
    {
        for (j=0; j<tab->size; j++)
        {
            if (tab->cont[i][j] != NULL)
            {
                DBlinkDestroy(tab->cont[i][j]);
            }
        }
        free(tab->cont[i]);
    }
    free(tab->cont);
    free(tab);

    return 0;
}
```

4.16.3.6 int DBlinkTabEnd (DBLinkTab ∗)

Definition at line 827 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), free, and DBLinkTab_::size.

Referenced by DBdestroy(), and DBnew().

```
{
    int i,j;

    if (tab == NULL || tab->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<tab->size;i++)
    {
        for (j=0; j<tab->size; j++)
        {
            if (tab->cont[i][j] != NULL)
            {
                DBlinkDestroy(tab->cont[i][j]);
            }
        }
        free(tab->cont[i]);
    }

    free(tab->cont);
    tab->cont = NULL;
    tab->size = 0;

    return 0;
}
```

4.16.3.7 int DBlinkTabInit (DBLinkTab ∗, long)

Definition at line 756 of file database_util.c.

References addError(), calloc, DBLinkTab_::cont, CRITICAL, free, LINKTAB_INITSIZE, and DBLinkTab_::size.

Referenced by DBnew().

```
{
    DBLink ***ptr=NULL;
    int i;

    if (tab == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LINKTAB_INITSIZE;

    if ((ptr=(calloc(size,sizeof(DBLink**))))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    else
    {
        for (i=0;i<size;i++)
        {
            if ((ptr[i]=(calloc(size, sizeof(DBLink*))))==NULL)
            {
                addError(CRITICAL,"Critical lack of memory in %s at line %d",
                         __FILE__,__LINE__);
                for (i=i-1;i>=0;i--)
                    free(ptr[i]);
                free(ptr);
                return -1;
            }
        }
    }

    tab->size=size;
    tab->cont=ptr;

    return 0;
}
```

4.16.3.8 DBLinkTab∗ DBlinkTabNew (long)

Definition at line 710 of file database_util.c.

References addError(), calloc, DBLinkTab_::cont, CRITICAL, free, LINKTAB_INITSIZE, and DBLinkTab_::size.

```
{
    DBLinkTab *tab=NULL;
    DBLink ***ptr=NULL;
    int i;

    if ((tab = calloc(1,sizeof(DBLinkTab))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LINKTAB_INITSIZE;

    if ((ptr=(calloc(size,sizeof(DBLink**))))==NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(tab);
        return NULL;
    }
    else
    {
        for (i=0;i<size;i++)
        {
            if ((ptr[i]=(calloc(size, sizeof(DBLink*))))==NULL)
            {
                addError(CRITICAL,"Critical lack of memory in %s at line %d",
                         __FILE__,__LINE__);
                for (i=i-1;i>=0;i--)
                    free(ptr[i]);
                free(ptr);
                free(tab);
                return NULL;
            }
        }
    }

    tab->size=size;
    tab->cont=ptr;

    return tab;
}
```

4.16.3.9 int DBlinkTabRemove (DBLinkTab *, long, long)

Definition at line 963 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), and DBLinkTab_::size.

Referenced by DBremoveLink().

```
{
    if (tab == NULL || tab->cont == NULL ||
        src <0 || dst<0 || src >= tab->size || dst >= tab->size ||
        tab->cont[src][dst] == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBlinkDestroy(tab->cont[src][dst]);
    tab->cont[src][dst]=NULL;

    return 0;
}
```

4.16.3.10 int DBlinkTabResize (DBLinkTab *, long)

Definition at line 858 of file database_util.c.

References addError(), calloc, DBLinkTab_::cont, CRITICAL, DBlinkDestroy(), free, min, realloc, and DBLinkTab_::size.

Referenced by DBlinkTabSet().

```
{
    DBLink*** ptr=NULL;
    DBLink** ptr2=NULL;
    int i,j;

    if (tab == NULL || tab->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size < tab->size)
    {
        for (i=size;i<tab->size;i++)
        {
            for (j=size; j<tab->size; j++)
            {
                if (tab->cont[i][j] != NULL)
                {
                    DBlinkDestroy(tab->cont[i][j]);
                    tab->cont[i][j]=NULL;
                }
            }
        }
        free(tab->cont[i]);
    }

    if ((ptr = realloc(tab->cont,size * sizeof(DBLink**))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    else
    {
        tab->cont = ptr;

        for (i=0;i<min(tab->size,size);i++)
        {
            if ((ptr2 = realloc(ptr[i], size * sizeof(DBLink*)))==NULL)
            {
                addError(CRITICAL,"Critical lack of memory in %s at line %d",
                         __FILE__,__LINE__);
                tab->size=min(tab->size,size);
                return -1;
            }

            ptr[i] = ptr2;

            if (size > tab->size)
            {
                memset(ptr2 + tab->size, 0, (size-tab->size) * sizeof(DBLink*));
            }
        }

        if (size > tab->size)
        {
            for (i=tab->size;i<size;i++)
            {
                if ((ptr[i] = calloc(size, sizeof(DBLink*)))==NULL)
                {
                    addError(CRITICAL,"Critical lack of memory in %s at line %d",
                             __FILE__,__LINE__);
                    tab->size=i;
                    return -1;
                }
            }
        }
    }

    tab->size=size;

    return 0;
}
```

4.16.3.11 int DBlinkTabSet (DBLinkTab *, DBLink *, long, long)

Definition at line 935 of file database_util.c.

References addError(), DBLinkTab_::cont, CRITICAL, DBlinkTabResize(), max, and DBLinkTab_::size.

Referenced by DBaddLink().

```
{
    long resize;

    if (tab == NULL || tab->cont == NULL || src <0 || dst<0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    resize=max(src,dst)+1;

    if (resize > tab->size)
    {
        if (DBlinkTabResize(tab,max(2*tab->size,resize))<0)
        {
            addError(CRITICAL,"Unable to resize link table prior to insertion in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    tab->cont[src][dst]=lnk;

    return 0;
}
```

4.16.3.12 int DBlspVecDestroy (DBLSPVec *)

Definition at line 565 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBlspDestroy(), free, and DBLSPVec_::size.

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBlspDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.16.3.13 int DBlspVecEnd (DBLSPVec *)

Definition at line 590 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBlspDestroy(), free, and DBLSPVec_::size.

Referenced by DBdestroy(), and DBnew().

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBlspDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;

    return 0;
}
```

4.16.3.14 int DBlspVecInit (DBLSPVec *, long)

Definition at line 538 of file database_util.c.

References addError(), calloc, DBLSPVec_::cont, CRITICAL, LSPVEC_INITSIZE, and DBLSPVec_-::size.

Referenced by DBnew().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = LSPVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->cont = ptr;

    return 0;
}
```

4.16.3.15 DBLSPVec* DBlspVecNew (long)

Definition at line 509 of file database_util.c.

References addError(), calloc, DBLSPVec_::cont, CRITICAL, free, LSPVEC_INITSIZE, and DBLSPVec_::size.

```
{
    DBLSPVec *vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(DBLSPVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = LSPVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->cont = ptr;

    return vec;
}
```

4.16.3.16 int DBlspVecRemove (DBLSPVec *, long)

Definition at line 689 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, and DBLSPVec_::size.

```
{
    if (vec == NULL || vec->cont == NULL ||
        id <0 || id >= vec->size || vec->cont[id] == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[id]=NULL;

    return 0;
}
```

4.16.3.17 int DBlspVecResize (DBLSPVec *, long)

Definition at line 616 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBLabelSwitchedPath, DBlspDestroy(), realloc, and DBLSPVec_::size.

Referenced by DBlspVecSet().

```
617 {
618     void *ptr=NULL;
619     int i;
620 
621     if (vec == NULL || vec->cont == NULL)
622     {
623         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
624                  __FILE__,__LINE__);
625         return -1;
626     }
627 
628     if (size < vec->size)
629     {
630         for (i=size;i<vec->size;i++)
631         {
632             if (vec->cont[i]!=NULL)
633             {
634                 DBlspDestroy(vec->cont[i]);
635                 vec->cont[i]=NULL;
636             }
637         }
638     }
639 
640     if ((ptr = realloc(vec->cont, size * sizeof(DBLabelSwitchedPath*))) == NULL)
641     {
642         addError(CRITICAL,"Critical lack of memory in %s at line %d",
643                  __FILE__,__LINE__);
644         return -1;
645     }
646 
647     if (size > vec->size)
648     {
649         memset(ptr + (vec->size * sizeof(DBLabelSwitchedPath*)), 0, (size-vec->size) * sizeof(DBLabelS
650     }
651 
652     vec->size=size;
653     vec->cont=ptr;
654 
655     return 0;
656 }
```

4.16.3.18 int DBlspVecSet (DBLSPVec *, DBLabelSwitchedPath *, long)

Definition at line 658 of file database_util.c.

References addError(), DBLSPVec_::cont, CRITICAL, DBlspVecResize(), max, and DBLSPVec_::size.

Referenced by DBaddLSP().

```
659 {
660     if (vec == NULL || vec->cont == NULL || id <0)
661     {
662         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
663                  __FILE__,__LINE__);
```

```
        return -1;
    }

    if (id >= vec->size)
    {
        if (DBlspVecResize(vec,max(2*vec->size,id+1))<0)
        {
            addError(CRITICAL,"Unable to resize LSP vector prior to insertion in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (vec->cont[id] != NULL)
    {
        addError(CRITICAL,"Trying to add an LSP with a reserved ID in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[id]=lsp;

    return 0;
}
```

4.16.3.19 int DBnodeDestroy (DBNode ∗)

clear function to free the ressources of a Link object allocated on the heap.

Definition at line 77 of file database_util.c.

References addError(), CRITICAL, free, DBNode_::inNeighb, longListEnd, and DBNode_::outNeighb.

Referenced by DBaddNode(), DBnodeVecDestroy(), DBnodeVecEnd(), DBnodeVecRemove(), and DBnodeVecResize().

```
{
    if (node == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    longListEnd(&(node->inNeighb));
    longListEnd(&(node->outNeighb));
    free(node);

    return 0;
}
```

4.16.3.20 int DBnodeEnd ()

4.16.3.21 int DBnodeInit (DBNode ∗)

initialize a DBNode object allready allocated somewhere else.

Definition at line 49 of file database_util.c.

References addError(), CRITICAL, DBNode_::inNeighb, longListEnd, longListInit, and DBNode_::outNeighb.

```
{
    if (node == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(node->inNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the incoming neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longListInit(&(node->outNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the outgoing neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&(node->inNeighb));
        return -1;
    }

    return 0;
}
```

4.16.3.22 DBNode∗ DBnodeNew ()

return a newly (dynamically) allocated DBNode object.

Definition at line 18 of file database_util.c.

References addError(), calloc, CRITICAL, free, DBNode_::inNeighb, longListEnd, longListInit, and DBNode_::outNeighb.

Referenced by DBaddNode().

```
{
    DBNode* ptr=NULL;

    if ((ptr = calloc(1,sizeof(DBNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(ptr->inNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the incoming neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (longListInit(&(ptr->outNeighb),-1) < 0)
    {
        addError(CRITICAL,"Unable to initialize the outgoing neighbour list in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&(ptr->inNeighb));
        free(ptr);
        return NULL;
    }

    return ptr;
}
```

4.16.3.23 int DBnodeVecDestroy (DBNodeVec ∗)

Definition at line 349 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeDestroy(), free, and DBNodeVec_::size.

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBnodeDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.16.3.24 int DBnodeVecEnd (DBNodeVec ∗)

Definition at line 374 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeDestroy(), free, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBdestroy(), and DBnew().

```
{
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0; i<vec->size; i++)
    {
        if (vec->cont[i]!=NULL)
        {
            DBnodeDestroy(vec->cont[i]);
        }
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.16.3.25 int DBnodeVecInit (DBNodeVec ∗, long)

Definition at line 321 of file database_util.c.

References addError(), calloc, DBNodeVec_::cont, CRITICAL, NODEVEC_INITSIZE, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBnew().

```
{
    void* ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = NODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBNode*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.16.3.26 DBNodeVec∗ DBnodeVecNew (long)

Definition at line 290 of file database_util.c.

References addError(), calloc, DBNodeVec_::cont, CRITICAL, free, NODEVEC_INITSIZE, DBNodeVec_::size, and DBNodeVec_::top.

```
{
    DBNodeVec *vec=NULL;
    void* ptr=NULL;

    if ((vec = calloc(1,sizeof(DBNodeVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = NODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(DBNode*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.16.3.27 int DBnodeVecRemove (DBNodeVec *, long)

Definition at line 485 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeDestroy(), DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBremoveNode().

```
{
    if (vec == NULL || vec->cont == NULL ||
        id <0 || id >= vec->size || vec->cont[id] == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    DBnodeDestroy(vec->cont[id]);
    vec->cont[id]=NULL;

    while (vec->cont[vec->top-1] == NULL)
        vec->top--;

    return 0;
}
```

4.16.3.28 int DBnodeVecResize (DBNodeVec *, long)

Definition at line 401 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBNode, DBnodeDestroy(), realloc, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBnodeVecSet().

```
{
    void *ptr=NULL;
    int i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size < vec->size)
    {
        for (i=size;i<vec->size;i++)
        {
            if (vec->cont[i]!=NULL)
            {
                DBnodeDestroy(vec->cont[i]);
                vec->cont[i]=NULL;
            }
        }

        if (size < vec->top)
        {
            vec->top = size;
            while (vec->cont[vec->top-1] == NULL)
                vec->top--;
        }

    }



    if ((ptr = realloc(vec->cont, size * sizeof(DBNode*))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size > vec->size)
    {
        memset(ptr + (vec->size * sizeof(DBNode*)), 0, (size-vec->size) * sizeof(DBNode*));
    }

    vec->size=size;
    vec->cont=ptr;

    return 0;
}
```

4.16.3.29 int DBnodeVecSet (DBNodeVec *, DBNode *, long)

Definition at line 453 of file database_util.c.

References addError(), DBNodeVec_::cont, CRITICAL, DBnodeVecResize(), max, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBaddNode().

```
{
    if (vec == NULL || vec->cont == NULL || node == NULL || id <0)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (id >= vec->size)
    {
        if (DBnodeVecResize(vec,max(2*vec->size,id+1))<0)
        {
            addError(CRITICAL,"Unable to resize node vector prior to insertion in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (vec->cont[id] != NULL)
    {
        addError(CRITICAL,"Trying to add a node with a reserved ID in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[id]=node;
    vec->top = max(vec->top, id+1);

    return 0;
}
```

4.16.3.30 void DBprintLink (DBLink *)

Definition at line 237 of file database_util.c.

References addError(), DBLinkState_::cap, DBLSPList_::cont, CRITICAL, DBLabelSwitchedPath_::id, DBLink_::lspList, NB_OA, NB_PREEMPTION, DBLinkState_::pbw, DBLinkState_::rbw, DBLink_::state, and DBLSPList_::top.

Referenced by DBprintDB().

```
{
    long i,oa;
    double ptot,rtot;
    DBLabelSwitchedPath* lsp=NULL;

    if (link == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return;
    }

    printf("\tList of LSPs\n");
    printf("\t------------\n");

    for (i=0; i<link->lspList.top; ++i)
    {
        lsp = link->lspList.cont[i];
        printf("%ld ", lsp->id);
    }

    printf("\n\n");

    printf("\tLink-state\n");
    printf("\t----------\n");

    for (oa=0; oa<NB_OA; ++oa)
    {
        ptot = 0;
        rtot = 0;

        printf("\tCapacity[%ld] = %f\n", oa, link->state.cap[oa]);

        for (i=0; i<NB_PREEMPTION; ++i)
        {
            ptot += link->state.pbw[oa][i];
            rtot += link->state.rbw[oa][i];
        }

        printf("\tpbw[%ld] = %f\n", oa, ptot);
        printf("\trbw[%ld] = %f\n", oa, rtot);
    }

    printf("\n\n");

}
```

4.16.3.31 void DBprintNode (DBNode *)

Definition at line 110 of file database_util.c.

References addError(), LongVec_::cont, CRITICAL, DBNode_::inNeighb, DBNode_::outNeighb, and LongVec_::top.

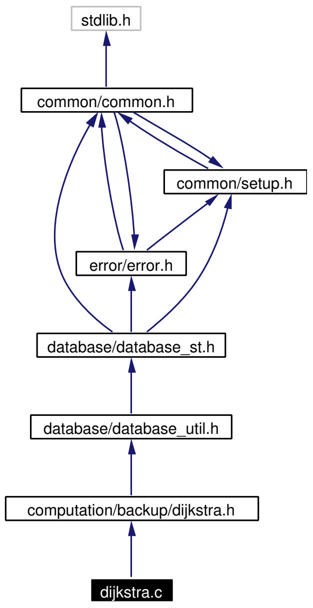
Referenced by DBprintDB().

```
{
    long i;

    if (node == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return;
    }

    printf("Incoming neighboors : \n");

    for (i=0; i<node->inNeighb.top; i++)
    {
        printf("%ld ", node->inNeighb.cont[i]);
    }

    printf("\nOutgoing neighboors : \n");

    for (i=0; i<node->outNeighb.top; i++)
    {
        printf("%ld ", node->outNeighb.cont[i]);
    }

    printf("\n");
}
```

4.17 dijkstra.c File Reference

```
#include "computation/backup/dijkstra.h"
```

Include dependency graph for dijkstra.c:



Functions

- CPTreeNode ∗ CPnewTN ()
- int CPdestroyTN (CPTreeNode ∗tn)
- int CPinitPQ (CPPrioQueue ∗pq)
- CPPrioQueue ∗ CPnewPQ ()
- int CPendPQ (CPPrioQueue ∗pq)
- int CPdestroyPQ (CPPrioQueue ∗pq)
- int CPinsertPQ (CPPrioQueue ∗pq, CPDijkNode ∗dn, double key)
- CPDijkNode ∗ CPpopTop (CPPrioQueue ∗pq)

4.17.1 Function Documentation

4.17.1.1 int CPdestroyPQ (CPPrioQueue ∗ *pq*)

Definition at line 87 of file dijkstra.c.

References addError(), CPendPQ(), CRITICAL, and free.

```
{
    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (CPendPQ(pq) < -1)
    {
        addError(CRITICAL,"Destruction incomplete in %s at line %d",
                 __FILE__,__LINE__);
         return -1;
     }

     free(pq);

     return 0;

}
```

4.17.1.2 int CPdestroyTN (CPTreeNode ∗ *tn*)

Definition at line 17 of file dijkstra.c.

References addError(), CRITICAL, and free.

Referenced by CPpopTop().

```
{
    if (tn == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(tn);

    return 0;
}
```

4.17.1.3 int CPendPQ (CPPrioQueue ∗ *pq*)

Definition at line 68 of file dijkstra.c.

References addError(), CPpopTop(), CRITICAL, CPPrioQueue_::root, CPPrioQueue_::size, and CPPrioQueue_::top.

Referenced by computeBackup(), and CPdestroyPQ().

```
{
    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    while ((CPpopTop(pq)) != NULL);

    pq->size = 0;
    pq->root = NULL;
    pq->top = NULL;

    return 0;

}
```

4.17.1.4 int CPinitPQ (CPPrioQueue ∗ *pq*)

Definition at line 32 of file dijkstra.c.

References addError(), CRITICAL, CPPrioQueue_::root, CPPrioQueue_::size, and CPPrioQueue_::top.

Referenced by computeBackup().

```
{
    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    pq->root = NULL;
    pq->top = NULL;
    pq->size = 0;

    return 0;
}
```

4.17.1.5 int CPinsertPQ (CPPrioQueue ∗ *pq*, CPDijkNode ∗ *dn*, double *key*)

Definition at line 109 of file dijkstra.c.

References addError(), CPnewTN(), CRITICAL, CPTreeNode_::father, CPTreeNode_::gt, CPTreeNode_::key, CPTreeNode_::leq, CPTreeNode_::node, CPPrioQueue_::root, CPPrioQueue_::size, and CPPrioQueue_::top.

Referenced by computeBackup().

```
{
    CPTreeNode* tn, *ptr, *lastPtr=NULL;

    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((tn = CPnewTN()) == NULL)
    {
        addError(CRITICAL,"Impossible to allocated a new TreeNode in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    tn->key = key;
    tn->node = dn;

    // size == 0
    if (pq->root == NULL)
    {
        pq->root = tn;
        pq->top = tn;
        pq->size++;
        return 0;
    }

    // lower than top
    if (key <= pq->top->key)
    {
        pq->top->leq = tn;
        tn->father = pq->top;
        pq->top = tn;
        pq->size++;
        return 0;
    }

    // anywhere else ...
    ptr = pq->root;

    while (ptr != NULL)
    {
        lastPtr = ptr;

        if (key <= ptr->key)
            ptr = ptr->leq;
        else
            ptr = ptr->gt;
    }

    if (key <= lastPtr->key)
        lastPtr->leq = tn;
    else
        lastPtr->gt = tn;

    tn->father = lastPtr;
    pq->size++;

    return 0;
}
```

4.17.1.6 CPPrioQueue∗ CPnewPQ ()

Definition at line 48 of file dijkstra.c.

References addError(), calloc, and CRITICAL.

```
{
    CPPrioQueue* pq=NULL;

    if ((pq = calloc(1, sizeof(CPPrioQueue))) == NULL)
    {
        addError(CRITICAL,"Impossible to allocated a new PrioQueue in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    /* Done by calloc !!!
    pq->root = NULL;
    pq->top = NULL;
    pq->size = 0;
    */

    return pq;
}
```

4.17.1.7 CPTreeNode∗ CPnewTN ()

Definition at line 3 of file dijkstra.c.

References addError(), calloc, and CRITICAL.

Referenced by CPinsertPQ().

```
{
    CPTreeNode* tn=NULL;

    if ((tn = calloc(1, sizeof(CPTreeNode))) == NULL)
    {
        addError(CRITICAL,"Impossible to allocated a new PrioQueue in %s at line %d",
                  __FILE__,__LINE__);
         return NULL;
     }

     return tn;
}
```

4.17.1.8 CPDijkNode∗ CPpopTop (CPPrioQueue ∗ *pq*)

Definition at line 173 of file dijkstra.c.

References addError(), CPdestroyTN(), CRITICAL, CPTreeNode_::father, CPTreeNode_::gt, CPTreeNode_::leq, CPTreeNode_::node, CPPrioQueue_::root, CPPrioQueue_::size, CPPrioQueue_::top, and WARNING.

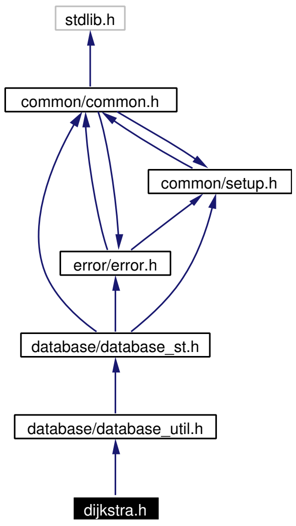
Referenced by computeBackup(), and CPendPQ().

```
{
    CPTreeNode* tn;
    CPDijkNode* dn;

    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    tn = pq->top;

    if (tn != NULL)
    {
        pq->size--;

        if (tn == pq->root)
        {
            pq->root = tn->gt;
            pq->top = tn->gt;
            if (tn->gt != NULL)
                tn->gt->father = NULL;
        }
        else
        {
            tn->father->leq = tn->gt;
            if (tn->gt != NULL)
            {
                pq->top = tn->gt;
                tn->gt->father = tn->father;
            }
            else
                pq->top = tn->father;
        }

        // now find the new top;
        if (pq->size > 0)
            while (pq->top->leq != NULL)
                pq->top = pq->top->leq;
    }
    else
    {
        return NULL;
    }

    dn = tn->node;

    if (CPdestroyTN(tn) < 0)
    {
        addError(WARNING,"Unable to destroy TreeNode but DijkNode was returned in %s at line %d",
                 __FILE__,__LINE__);
    }

    return dn;
}
```

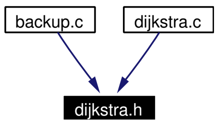
4.18 dijkstra.h File Reference

```
#include "database/database_util.h"
```

Include dependency graph for dijkstra.h:



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



Data Structures

- struct CPDijkNode_
- struct CPPrioQueue_
- struct CPTreeNode_

Typedefs

- typedef CPDijkNode_ CPDijkNode
- typedef CPTreeNode_ CPTreeNode
- typedef CPPrioQueue_ CPPrioQueue

Functions

- int CPdestroyTN (CPTreeNode ∗)
- int CPinitPQ (CPPrioQueue ∗)
- CPPrioQueue ∗ CPnewPQ ()

- int CPendPQ (CPPrioQueue ∗)
- int CPdestroyPQ (CPPrioQueue ∗)
- int CPinsertPQ (CPPrioQueue ∗, CPDijkNode ∗, double)
- CPDijkNode ∗ CPpopTop (CPPrioQueue ∗)

4.18.1 Typedef Documentation

4.18.1.1 typedef struct CPDijkNode_ CPDijkNode

4.18.1.2 typedef struct CPPrioQueue_ CPPrioQueue

4.18.1.3 typedef struct CPTreeNode_ CPTreeNode

4.18.2 Function Documentation

4.18.2.1 int CPdestroyPQ (CPPrioQueue ∗)

Definition at line 87 of file dijkstra.c.

References addError(), CPendPQ(), CRITICAL, and free.

```
{
    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (CPendPQ(pq) < -1)
    {
        addError(CRITICAL,"Destruction incomplete in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(pq);

    return 0;

}
```

4.18.2.2 int CPdestroyTN (CPTreeNode ∗)

Definition at line 17 of file dijkstra.c.

References addError(), CRITICAL, and free.

Referenced by CPpopTop().

```
{
    if (tn == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(tn);

    return 0;
}
```

4.18.2.3 int CPendPQ (CPPrioQueue ∗)

Definition at line 68 of file dijkstra.c.

References addError(), CPpopTop(), CRITICAL, CPPrioQueue_::root, CPPrioQueue_::size, and CPPrioQueue_::top.

Referenced by computeBackup(), and CPdestroyPQ().

```
{
    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    while ((CPpopTop(pq)) != NULL);

    pq->size = 0;
    pq->root = NULL;
    pq->top = NULL;

    return 0;

}
```

4.18.2.4 int CPinitPQ (CPPrioQueue ∗)

Definition at line 32 of file dijkstra.c.

References addError(), CRITICAL, CPPrioQueue_::root, CPPrioQueue_::size, and CPPrioQueue_::top.

Referenced by computeBackup().

```
{
    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    pq->root = NULL;
    pq->top = NULL;
    pq->size = 0;

    return 0;
}
```

4.18.2.5 int CPinsertPQ (CPPrioQueue ∗, CPDijkNode ∗, double)

Definition at line 109 of file dijkstra.c.

References addError(), CPnewTN(), CRITICAL, CPTreeNode_::father, CPTreeNode_::gt, CPTreeNode_::key, CPTreeNode_::leq, CPTreeNode_::node, CPPrioQueue_::root, CPPrioQueue_::size, and CPPrioQueue_::top.

Referenced by computeBackup().

```
{
    CPTreeNode* tn, *ptr, *lastPtr=NULL;

    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((tn = CPnewTN()) == NULL)
    {
        addError(CRITICAL,"Impossible to allocated a new TreeNode in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    tn->key = key;
    tn->node = dn;

    // size == 0
    if (pq->root == NULL)
    {
        pq->root = tn;
        pq->top = tn;
        pq->size++;
        return 0;
    }

    // lower than top
    if (key <= pq->top->key)
    {
        pq->top->leq = tn;
        tn->father = pq->top;
        pq->top = tn;
        pq->size++;
        return 0;
    }

    // anywhere else ...
    ptr = pq->root;

    while (ptr != NULL)
    {
        lastPtr = ptr;

        if (key <= ptr->key)
            ptr = ptr->leq;
        else
            ptr = ptr->gt;
    }

    if (key <= lastPtr->key)
        lastPtr->leq = tn;
    else
        lastPtr->gt = tn;

    tn->father = lastPtr;
    pq->size++;

    return 0;
}
```

4.18.2.6 CPPrioQueue* CPnewPQ ()

Definition at line 48 of file dijkstra.c.

References addError(), calloc, and CRITICAL.

```
{
    CPPrioQueue* pq=NULL;

    if ((pq = calloc(1, sizeof(CPPrioQueue))) == NULL)
    {
        addError(CRITICAL,"Impossible to allocated a new PrioQueue in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    /* Done by calloc !!!
    pq->root = NULL;
    pq->top = NULL;
    pq->size = 0;
    */

    return pq;
}
```

4.18.2.7 CPDijkNode* CPpopTop (CPPrioQueue *)

Definition at line 173 of file dijkstra.c.

References addError(), CPdestroyTN(), CRITICAL, CPTreeNode_::father, CPTreeNode_::gt, CPTreeNode_::leq, CPTreeNode_::node, CPPrioQueue_::root, CPPrioQueue_::size, CPPrioQueue_::top, and WARNING.

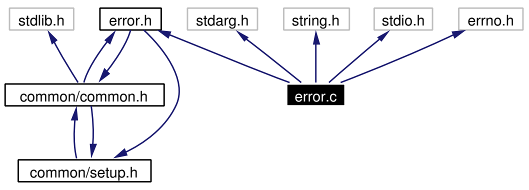
Referenced by computeBackup(), and CPendPQ().

```
{
    CPTreeNode* tn;
    CPDijkNode* dn;

    if (pq == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    tn = pq->top;

    if (tn != NULL)
    {
        pq->size--;

        if (tn == pq->root)
        {
            pq->root = tn->gt;
            pq->top = tn->gt;
            if (tn->gt != NULL)
                tn->gt->father = NULL;
        }
        else
        {
            tn->father->leq = tn->gt;
            if (tn->gt != NULL)
            {
                pq->top = tn->gt;
                tn->gt->father = tn->father;
            }
            else
                pq->top = tn->father;
        }

        // now find the new top;
        if (pq->size > 0)
            while (pq->top->leq != NULL)
                pq->top = pq->top->leq;
    }
    else
    {
        return NULL;
    }

    dn = tn->node;

    if (CPdestroyTN(tn) < 0)
    {
        addError(WARNING,"Unable to destroy TreeNode but DijkNode was returned in %s at line %d",
                 __FILE__,__LINE__);
    }

    return dn;
}
```

4.19 error.c File Reference

```
#include "error.h"
#include <stdio.h>
#include <errno.h>
#include <stdarg.h>
#include <string.h>
```

Include dependency graph for error.c:



Data Structures

- struct ErrorElem_
- struct ErrorList_

Typedefs

- typedef ErrorElem_ ErrorElem

Functions

- void errorInit ()
- void addError (GravityLevel level, const char ∗msg,...)
- void printErrorStack ()
- void errorDestroy ()

Variables

- ErrorList errorManager

4.19.1 Typedef Documentation

4.19.1.1 typedef struct ErrorElem_ ErrorElem

4.19.2 Function Documentation

4.19.2.1 void addError (GravityLevel *level*, const char ∗ *msg*, ...)

Definition at line 40 of file error.c.

References CRITICAL, ERROR_PROVISION, errorManager, ERRORMSG_SIZE, INFO, ErrorList_::list, PANIC, realloc, ErrorList_::size, ErrorList_::top, and WARNING.

Referenced by activateNodeInfo(), bellmanKalaba(), bkConnectVecCopy(), bkConnectVecDestroy(), bkConnectVecEnd(), bkConnectVecGet(), bkConnectVecInit(), bkConnectVecPopBack(), bkConnectVecPushBack(), bkConnectVecResize(), bkConnectVecSet(), bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecGet(), bkNodeVecInit(), bkNodeVecNew(), bkNodeVecPopBack(), bkNodeVecPushBack(), bkNodeVecResize(), bkNodeVecSet(), chooseReroutedLSPs(), computeBackup(), computeCost(), computePrimaryPath(), computeRBW(), CPdestroyPQ(), CPdestroyTN(), CPendPQ(), CPinitPQ(), CPinsertPQ(), CPnewPQ(), CPnewTN(), CPpopTop(), DBaddLink(), DBaddLSP(), DBaddNode(), DBdestroy(), DBgetID(), DBgetLinkDst(), DBgetLinkID(), DBgetLinkLSPs(), DBgetLinkSrc(), DBgetLinkState(), DBgetLSP(), DBgetMaxNodeID(), DBgetNbLinks(), DBgetNbNodes(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), DBlinkDestroy(), DBlinkEnd(), DBlinkInit(), DBlinkNew(), DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), DBlinkStateNew(), DBlinkTabDestroy(), DBlinkTabEnd(), DBlinkTabInit(), DBlinkTabNew(), DBlinkTabRemove(), DBlinkTabResize(), DBlinkTabSet(), DBlspCopy(), DBlspDestroy(), DBlspEnd(), DBlspInit(), DBlspListDestroy(), DBlspListEnd(), DBlspListInit(), DBlspListInsert(), DBlspListNew(), DBlspListRemove(), DBlspNew(), DBlspVecDestroy(), DBlspVecEnd(), DBlspVecInit(), DBlspVecNew(), DBlspVecRemove(), DBlspVecResize(), DBlspVecSet(), dblVecCopy(), dblVecDestroy(), dblVecEnd(), dblVecGet(), dblVecInit(), dblVecNew(), dblVecPopBack(), dblVecPushBack(), dblVecResize(), dblVecSet(), DBnew(), DBnodeDestroy(), DBnodeEnd(), DBnodeInit(), DBnodeNew(), DBnodeVecDestroy(), DBnodeVecEnd(), DBnodeVecInit(), DBnodeVecNew(), DBnodeVecRemove(), DBnodeVecResize(), DBnodeVecSet(), DBprintLink(), DBprintNode(), DBremoveLink(), DBremoveLSP(), DBremoveNode(), DBsetLinkState(), endTopo(), evalLS(), fillTopo(), getRequestDst(), getRequestSrc(), initScore(), initTopo(), isValidLSPLink(), isValidRequestLink(), longListInsert(), longListMerge(), longListRemove(), longListSort(), longVecCopy(), longVecDestroy(), longVecEnd(), longVecGet(), longVecInit(), longVecNew(), longVecPopBack(), longVecPushBack(), longVecResize(), longVecSet(), lspRequestCopy(), lspRequestDestroy(), lspRequestEnd(), lspRequestInit(), lspRequestListEnd(), lspRequestListGet(), lspRequestListInit(), lspRequestListResize(), lspRequestListSet(), lspRequestListSize(), lspRequestNew(), makeRerouteScore(), makeScore(), printTopo(), updateLS(), updateNodeInfoOnElect(), and updateRequest().

```
{
    va_list lst;
    void *ptr=NULL;
    char tmpmsg[ERRORMSG_SIZE];

    va_start(lst,msg);

    vsnprintf(tmpmsg,ERRORMSG_SIZE,msg,lst);
    tmpmsg[ERRORMSG_SIZE-1]='\0';


    switch (level)
    {
        case INFO:
        case WARNING:
        case CRITICAL:
        case PANIC:
            break;
    }

    if (errorManager.top >= errorManager.size-ERROR_PROVISION)
    {
        if (( ptr = realloc(errorManager.list, errorManager.size *
                            2 * sizeof(ErrorElem))) == NULL)
        {
            if (errorManager.top < errorManager.size)
            {
                errorManager.list[errorManager.top].gravity = CRITICAL;
                strncpy(errorManager.list[errorManager.top].message,
                        "Critical lack of memory encountered while resizing error manager",
                        ERRORMSG_SIZE);
                errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
            }
            else
                addError(PANIC,"");

            if (errorManager.top < errorManager.size)
            {
                errorManager.list[errorManager.top].gravity = level;
                strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
                errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
            }
        }
        else
        {
            errorManager.list=ptr;
            errorManager.size*=2;
            errorManager.list[errorManager.top].gravity = level;
            strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
            errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
        }
    }
    else
    {
        errorManager.list[errorManager.top].gravity = level;
        strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
        errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
    }
}
```

4.19.2.2 void errorDestroy ()

Definition at line 131 of file error.c.

References errorManager, free, ErrorList_::list, ErrorList_::size, and ErrorList_::top.

```
{
    free(errorManager.list);
    errorManager.top=0;
    errorManager.size=0;
}
```

4.19.2.3 void errorInit ()

Definition at line 29 of file error.c.

References calloc, ERRORLIST_INITSIZE, errorManager, ErrorList_::list, ErrorList_::size, and Error-List_::top.

```
{
    if ((errorManager.list=calloc(ERRORLIST_INITSIZE,sizeof(ErrorElem)))==NULL)
    {
        perror("Unable to initialize error manager");
        abort();
    }
    errorManager.top=0;
    errorManager.size=ERRORLIST_INITSIZE;
}
```

4.19.2.4 void printErrorStack ()

Definition at line 101 of file error.c.

References CRITICAL, errorManager, ErrorElem_::gravity, INFO, ErrorList_::list, ErrorElem_::message, PANIC, ErrorList_::top, and WARNING.

```
{
    long i;
    ErrorElem error;

    for (i=errorManager.top-1; i>=0; i--)
    {
        error = errorManager.list[i];

        switch(error.gravity)
        {
            case INFO:
                printf("[INFO] ");
                break;
            case WARNING:
                printf("[WARNING] ");
                break;
            case CRITICAL:
                printf("[CRITICAL] ");
                break;
            case PANIC:
                printf("[PANIC] ");

        }

        printf("%s\n", error.message);

    }
}
```

4.19.3 Variable Documentation

4.19.3.1 ErrorList errorManager

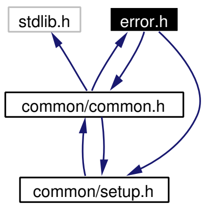
Definition at line 26 of file error.c.

Referenced by addError(), errorDestroy(), errorInit(), and printErrorStack().

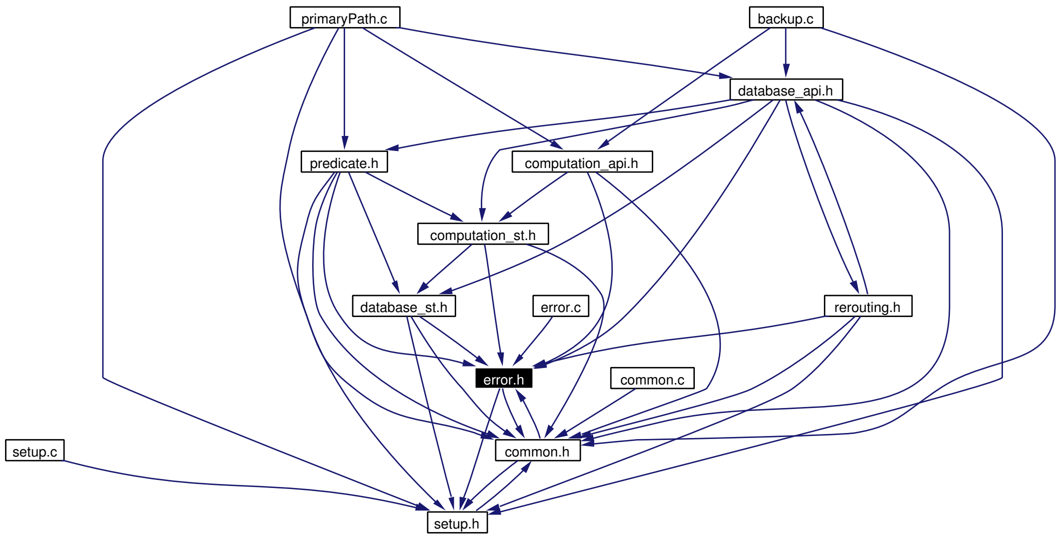
4.20 error.h File Reference

```
#include "common/common.h"
#include "common/setup.h"
```

Include dependency graph for error.h:



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



Typedefs

- typedef ErrorList_ ErrorList

Enumerations

- enum GravityLevel { INFO, WARNING, CRITICAL, PANIC }

Functions

- void errorInit ()
- void errorDestroy ()
- void addError (GravityLevel, const char ∗,...)
- void printErrorStack ()

Variables

- ErrorList errorManager

4.20.1 Typedef Documentation

4.20.1.1 typedef struct ErrorList_ ErrorList

Definition at line 18 of file error.h.

4.20.2 Enumeration Type Documentation

4.20.2.1 enum GravityLevel

Enumeration values:

INFO

WARNING

CRITICAL

PANIC

Definition at line 11 of file error.h.

```
11 {INFO,WARNING,CRITICAL,PANIC} GravityLevel;
```

4.20.3 Function Documentation

4.20.3.1 void addError (GravityLevel, const char *, ...)

Definition at line 40 of file error.c.

References CRITICAL, ERROR_PROVISION, errorManager, ERRORMSG_SIZE, INFO, ErrorList_::list, PANIC, realloc, ErrorList_::size, ErrorList_::top, and WARNING.

Referenced by activateNodeInfo(), bellmanKalaba(), bkConnectVecCopy(), bkConnectVecDestroy(), bkConnectVecEnd(), bkConnectVecGet(), bkConnectVecInit(), bkConnectVecPopBack(), bkConnectVecPushBack(), bkConnectVecResize(), bkConnectVecSet(), bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecGet(), bkNodeVecInit(), bkNodeVecNew(), bkNodeVecPopBack(), bkNodeVecPushBack(), bkNodeVecResize(), bkNodeVecSet(), chooseReroutedLSPs(), computeBackup(), computeCost(), computePrimaryPath(), computeRBW(), CPdestroyPQ(), CPdestroyTN(), CPendPQ(), CPinitPQ(), CPinsertPQ(), CPnewPQ(), CPnewTN(), CPpopTop(), DBaddLink(), DBaddLSP(), DBaddNode(), DBdestroy(), DBgetID(), DBgetLinkDst(), DBgetLinkID(), DBgetLinkLSPs(), DBgetLinkSrc(), DBgetLinkState(), DBgetLSP(), DBgetMaxNodeID(), DBgetNbLinks(), DBgetNbNodes(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), DBlinkDestroy(), DBlinkEnd(), DBlinkInit(), DBlinkNew(), DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), DBlinkStateNew(), DBlinkTabDestroy(), DBlinkTabEnd(), DBlinkTabInit(), DBlinkTabNew(), DBlinkTabRemove(), DBlinkTabResize(), DBlinkTabSet(), DBlspCopy(), DBlspDestroy(), DBlspEnd(), DBlspInit(), DBlspListDestroy(), DBlspListEnd(), DBlspListInit(), DBlspListInsert(), DBlspListNew(), DBlspListRemove(), DBlspNew(), DBlspVecDestroy(), DBlspVecEnd(), DBlspVecInit(), DBlspVecNew(), DBlspVecRemove(), DBlspVecResize(), DBlspVecSet(), dblVecCopy(), dblVecDestroy(), dblVecEnd(), dblVecGet(), dblVecInit(), dblVecNew(), dblVecPopBack(), dblVecPushBack(), dblVecResize(), dblVecSet(), DBnew(), DBnodeDestroy(), DBnodeEnd(), DBnodeInit(), DBnodeNew(), DBnodeVecDestroy(), DBnodeVecEnd(), DBnodeVecInit(), DBnodeVecNew(), DBnodeVecRemove(), DBnodeVecResize(), DBnodeVecSet(), DBprintLink(), DBprintNode(), DBremoveLink(), DBremoveLSP(), DBremoveNode(), DBsetLinkState(), endTopo(), evalLS(), fillTopo(), getRequestDst(), getRequestSrc(), initScore(), initTopo(), isValidLSPLink(), isValidRequestLink(), longListInsert(), longListMerge(), longListRemove(), longListSort(), longVecCopy(), longVecDestroy(), longVecEnd(), longVecGet(), longVecInit(), longVecNew(), longVecPopBack(), longVecPushBack(), longVecResize(), longVecSet(), lspRequestCopy(), lspRequestDestroy(), lspRequestEnd(), lspRequestInit(), lspRequestListEnd(), lspRequestListGet(), lspRequestListInit(), lspRequestListResize(), lspRequestListSet(), lspRequestListSize(), lspRequestNew(), makeRerouteScore(), makeScore(), printTopo(), updateLS(), updateNodeInfoOnElect(), and updateRequest().

```
{
    va_list lst;
    void *ptr=NULL;
    char tmpmsg[ERRORMSG_SIZE];

    va_start(lst,msg);

    vsnprintf(tmpmsg,ERRORMSG_SIZE,msg,lst);
    tmpmsg[ERRORMSG_SIZE-1]='\0';


    switch (level)
    {
        case INFO:
        case WARNING:
        case CRITICAL:
        case PANIC:
            break;
    }

    if (errorManager.top >= errorManager.size-ERROR_PROVISION)
    {
        if (( ptr = realloc(errorManager.list, errorManager.size *
                            2 * sizeof(ErrorElem))) == NULL)
        {
            if (errorManager.top < errorManager.size)
            {
                errorManager.list[errorManager.top].gravity = CRITICAL;
                strncpy(errorManager.list[errorManager.top].message,
                        "Critical lack of memory encountered while resizing error manager",
                        ERRORMSG_SIZE);
                errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
            }
            else
                addError(PANIC,"");

            if (errorManager.top < errorManager.size)
            {
                errorManager.list[errorManager.top].gravity = level;
                strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
                errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
            }
        }
        else
        {
            errorManager.list=ptr;
            errorManager.size*=2;
            errorManager.list[errorManager.top].gravity = level;
            strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
            errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
        }
    }
    else
    {
```

```
95          errorManager.list[errorManager.top].gravity = level;
96          strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
97          errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
98      }
99 }
```

4.20.3.2 void errorDestroy ()

Definition at line 131 of file error.c.

References errorManager, free, ErrorList_::list, ErrorList_::size, and ErrorList_::top.

```
132 {
133     free(errorManager.list);
134     errorManager.top=0;
135     errorManager.size=0;
136 }
```

4.20.3.3 void errorInit ()

Definition at line 29 of file error.c.

References calloc, ERRORLIST_INITSIZE, errorManager, ErrorList_::list, ErrorList_::size, and ErrorList_::top.

```
30 {
31     if ((errorManager.list=calloc(ERRORLIST_INITSIZE,sizeof(ErrorElem)))==NULL)
32     {
33         perror("Unable to initialize error manager");
34         abort();
35     }
36     errorManager.top=0;
37     errorManager.size=ERRORLIST_INITSIZE;
38 }
```

4.20.3.4 void printErrorStack ()

Definition at line 101 of file error.c.

References CRITICAL, errorManager, ErrorElem_::gravity, INFO, ErrorList_::list, ErrorElem_::message, PANIC, ErrorList_::top, and WARNING.

```
102 {
103     long i;
104     ErrorElem error;
105 
106     for (i=errorManager.top-1; i>=0; i--)
107     {
108         error = errorManager.list[i];
109 
110         switch(error.gravity)
111         {
112             case INFO:
113                 printf("[INFO] ");
114                 break;
115             case WARNING:
116                 printf("[WARNING] ");
```

```
                break;
            case CRITICAL:
                printf("[CRITICAL] ");
                break;
            case PANIC:
                printf("[PANIC] ");

        }

        printf("%s\n", error.message);

    }
}
```

4.20.4 Variable Documentation

4.20.4.1 ErrorList errorManager

Definition at line 19 of file error.h.

Referenced by addError(), errorDestroy(), errorInit(), and printErrorStack().

4.21 predicate.c File Reference

`#include "predicate.h"`

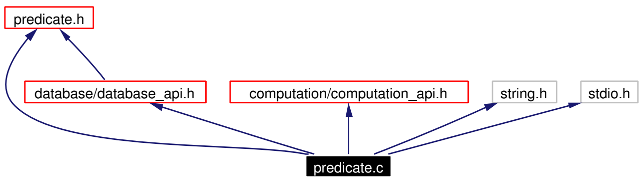
`#include "computation/computation_api.h"`

`#include "database/database_api.h"`

`#include <string.h>`

`#include <stdio.h>`

Include dependency graph for predicate.c:



Functions

- bool capacityClause (DBLinkState ∗ls, LSPRequest ∗req, double gain[NB_OA])
- bool colorClause (DBLinkState ∗ls, LSPRequest ∗req)
- bool isValidRequestLink (DataBase ∗dataBase, long src, long dst, DBLinkState ∗ls, LSPRequest ∗req, double gain[NB_OA])
- bool isValidLSPLink (DataBase ∗dataBase, long src, long dst, DBLinkState ∗ls, DBLabelSwitchedPath ∗lsp, double gain[NB_OA])

4.21.1 Function Documentation

4.21.1.1 bool capacityClause (DBLinkState ∗ *ls*, LSPRequest ∗ *req*, double *gain*[NB_OA])

Definition at line 7 of file predicate.c.

References PredicateConfig_::allowReroute, DBLinkState_::cap, damoteConfig, FALSE, NB_OA, NB_PREEMPTION, LSPRequest_::precedence, DAMOTEConfig_::predicateConfig, DBLinkState_::rbw, and TRUE.

Referenced by isValidRequestLink().

```
8 {
9     double occupied[NB_OA],total[NB_OA];
10     int i,j;
11
12     for (i=0;i<NB_OA;i++)
13     {
14         occupied[i]=0;
15         total[i]=0;
16         for (j=0;j<NB_PREEMPTION;j++)
17         {
18             if (j<=(damoteConfig.predicateConfig.allowReroute?req->precedence:NB_PREEMPTION))
19                 occupied[i]+=ls->rbw[i][j];
20             total[i]+=ls->rbw[i][j];
21         }
22     }
```

```

    for (i=0;i<NB_OA;i++)
    {
        if (ls->cap[i]<occupied[i])
            return FALSE;
        if (ls->cap[i]<total[i])
            gain[i]=total[i]-ls->cap[i];
        else
            gain[i]=0;
    }

    return TRUE;
}
```

4.21.1.2 bool colorClause (DBLinkState ∗ *ls*, LSPRequest ∗ *req*)

Definition at line 37 of file predicate.c.

References DBLinkState_::color, LongVec_::cont, FALSE, LSPRequest_::forbidLinks, LongVec_::top, and TRUE.

Referenced by isValidRequestLink().

```
{
    int i;

    for (i=0;i<req->forbidLinks.top;i++)
    {
        if (req->forbidLinks.cont[i]==ls->color)
            return FALSE;
    }

    return TRUE;
}
```

4.21.1.3 bool isValidLSPLink (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *ls*, DBLabelSwitchedPath ∗ *lsp*, double *gain*[NB_OA])

Definition at line 169 of file predicate.c.

References addError(), LSPRequest_::bw, DBLabelSwitchedPath_::bw, CRITICAL, LSPrerouteInfo_::dst, FALSE, LSPRequest_::forbidLinks, DBLabelSwitchedPath_::forbidLinks, LSPRequest_::id, DBLabelSwitchedPath_::id, LSPrerouteInfo_::id, isValidRequestLink(), longListCopy, lspRequestEnd(), lspRequestInit(), NB_OA, DBLabelSwitchedPath_::noContentionId, LSPRequest_::path, DBLabelSwitchedPath_::path, LSPRequest_::precedence, DBLabelSwitchedPath_::precedence, LSPRequest_::primID, DBLabelSwitchedPath_::primID, LSPRequest_::rerouteInfo, LSPrerouteInfo_::src, LSPRequest_::type, and DBLabelSwitchedPath_::type.

Referenced by DBaddLSP().

```
{
    LSPRequest req;
    bool gate;

    if (ls == NULL || lsp == NULL || gain==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    if (lspRequestInit(&req)<0)
    {
        addError(CRITICAL,"Unable to initialize request in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    req.id=lsp->id;
    req.precedence=lsp->precedence;
    req.type=lsp->type;
    req.primID=lsp->primID;
    req.rerouteInfo.id=lsp->noContentionId;
    req.rerouteInfo.src=-1;
    req.rerouteInfo.dst=-1;
    memcpy(req.bw,lsp->bw, NB_OA * sizeof(double));

    if (longListCopy(&(req.forbidLinks),&(lsp->forbidLinks))<0)
    {
        addError(CRITICAL,"Unable to initialize request in %s at line %d",
                 __FILE__,__LINE__);
        lspRequestEnd(&req);
        return FALSE;
    }

    if (longListCopy(&(req.path),&(lsp->path))<0)
    {
        addError(CRITICAL,"Unable to initialize request in %s at line %d",
                 __FILE__,__LINE__);
        lspRequestEnd(&req);
        return FALSE;
    }


    gate=isValidRequestLink(dataBase,src,dst,ls,&req,gain);

    lspRequestEnd(&req);

    return gate;
}
```

4.21.1.4 bool isValidRequestLink (DataBase ∗ *dataBase*, long *src*, long *dst*, DBLinkState ∗ *ls*, LSPRequest ∗ *req*, double *gain*[NB_OA])

Definition at line 50 of file predicate.c.

References addError(), PredicateConfig_::capacityClause, capacityClause(), PredicateConfig_::colorClause, colorClause(), LongVec_::cont, CRITICAL, damoteConfig, DBevalLSOnSetup(), DBlinkStateEnd(), DBlinkStateInit(), LSPrerouteInfo_::dst, FALSE, LSPRequest_::id, LSPrerouteInfo_::id, longListCopy, longListEnd, longListInit, longListPushBack, LSPRequest_::path, DAMOTEConfig_::predicateConfig, LSPRequest_::rerouteInfo, LSPrerouteInfo_::src, LongVec_::top, and TRUE.

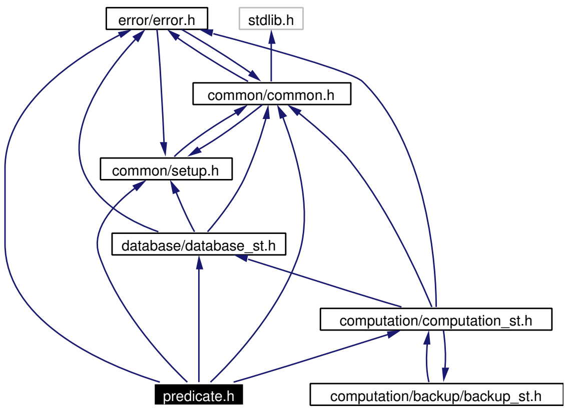
Referenced by fillTopo(), and isValidLSPLink().

```
{
    if (req->id == 1781) {
        int olivier;
        fprintf(stderr,"\nbackup %ld: ", req->id);
        for (olivier=0;olivier<req->path.top;olivier++){
            fprintf(stderr,"%ld ", req->path.cont[olivier]);
        }
    }

    LongList recPath;
    DBLinkState newLS;
    bool gate=TRUE,tmpgate,completePath=TRUE;
    long i;

    if (ls == NULL || req == NULL || gain==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    if (DBlinkStateInit(&newLS)<0)
    {
        addError(CRITICAL,"Unable to initialize link state in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    for (i=0;i<req->path.top && completePath;i++)
    {
        if (req->path.cont[i]<0)
        {
            completePath=FALSE;
        }
    }

    if (! completePath)
    {
        if (longListInit(&recPath,-1)<0)
        {
            addError(CRITICAL,"Unable to initialize path record structure in %s at line %d",
                     __FILE__,__LINE__);
            return FALSE;
        }
        if (longListCopy(&recPath,&(req->path))<0)
        {
            addError(CRITICAL,"Unable to copy path into record structure in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&recPath);
            return FALSE;
        }
        req->path.top=0;
        if (longListPushBack(&(req->path),src)<0 ||
            longListPushBack(&(req->path),dst)<0)
        {
            addError(CRITICAL,"Unable to forge path into request in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&recPath);
            return FALSE;
        }
    }

    if (req->id == 1781) {
        int olivier;
        fprintf(stderr,"\nbackup %ld: ", req->id);
        for (olivier=0;olivier<req->path.top;olivier++){
            fprintf(stderr,"%ld ", req->path.cont[olivier]);
        }
    }


    if (DBevalLSOnSetup(dataBase,src,dst,&newLS,ls,req)<0)
    {
        addError(CRITICAL,"Unable to update link state in %s at line %d",
                 __FILE__,__LINE__);
        if (!completePath)
        {
            longListEnd(&recPath);
        }
        return FALSE;
    }

    if (damoteConfig.predicateConfig.capacityClause)
    {
        tmpgate=capacityClause(&newLS,req,gain);
        gate=gate && tmpgate;
    }

    if (damoteConfig.predicateConfig.colorClause)
    {
        tmpgate=colorClause(&newLS,req);
        gate=gate && tmpgate;
    }

    if (req->rerouteInfo.id >= 0 && req->rerouteInfo.src==src
        && req->rerouteInfo.dst==dst)
    {
        gate= FALSE;
    }

    if (!completePath)
    {
        if (longListCopy(&(req->path),&recPath)<0)
        {
            addError(CRITICAL,"Unable to restore path into request in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&recPath);
            return FALSE;
        }
        longListEnd(&recPath);
    }

    DBlinkStateEnd(&newLS);

    return gate;
}
```

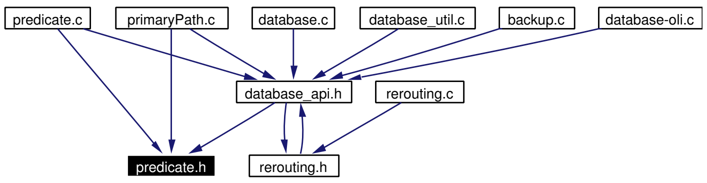
4.22 predicate.h File Reference

```
#include "error/error.h"
#include "computation/computation_st.h"
#include "common/common.h"
#include "common/setup.h"
#include "database/database_st.h"
```

Include dependency graph for predicate.h:



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



Functions

- bool isValidRequestLink (DataBase ∗, long, long, DBLinkState ∗, LSPRequest ∗, double[NB_OA])
- bool isValidLSPLink (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗, double[NB_OA])

4.22.1 Function Documentation

4.22.1.1 bool isValidLSPLink (DataBase ∗, long, long, DBLinkState ∗, DBLabelSwitchedPath ∗, double[NB_OA])

Definition at line 169 of file predicate.c.

References addError(), DBLabelSwitchedPath_::bw, LSPRequest_::bw, CRITICAL, LSPrerouteInfo_::dst, FALSE, DBLabelSwitchedPath_::forbidLinks, LSPRequest_::forbidLinks, LSPrerouteInfo_::id, DBLabel-

SwitchedPath_::id, LSPRequest_::id, isValidRequestLink(), longListCopy, lspRequestEnd(), lspRequestInit(), NB_OA, DBLabelSwitchedPath_::noContentionId, DBLabelSwitchedPath_::path, LSPRequest_::path, DBLabelSwitchedPath_::precedence, LSPRequest_::precedence, DBLabelSwitchedPath_::primID, LSPRequest_::primID, LSPRequest_::rerouteInfo, LSPrerouteInfo_::src, DBLabelSwitchedPath_::type, and LSPRequest_::type.

Referenced by DBaddLSP().

```
{
    LSPRequest req;
    bool gate;

    if (ls == NULL || lsp == NULL || gain==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    if (lspRequestInit(&req)<0)
    {
        addError(CRITICAL,"Unable to initialize request in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    req.id=lsp->id;
    req.precedence=lsp->precedence;
    req.type=lsp->type;
    req.primID=lsp->primID;
    req.rerouteInfo.id=lsp->noContentionId;
    req.rerouteInfo.src=-1;
    req.rerouteInfo.dst=-1;
    memcpy(req.bw,lsp->bw, NB_OA * sizeof(double));

    if (longListCopy(&(req.forbidLinks),&(lsp->forbidLinks))<0)
    {
        addError(CRITICAL,"Unable to initialize request in %s at line %d",
                 __FILE__,__LINE__);
        lspRequestEnd(&req);
        return FALSE;
    }

    if (longListCopy(&(req.path),&(lsp->path))<0)
    {
        addError(CRITICAL,"Unable to initialize request in %s at line %d",
                 __FILE__,__LINE__);
        lspRequestEnd(&req);
        return FALSE;
    }


    gate=isValidRequestLink(dataBase,src,dst,ls,&req,gain);

    lspRequestEnd(&req);

    return gate;
}
```

4.22.1.2 bool isValidRequestLink (DataBase ∗, long, long, DBLinkState ∗, LSPRequest ∗, double[NB_OA])

Definition at line 50 of file predicate.c.

References addError(), capacityClause(), PredicateConfig_::capacityClause, colorClause(), PredicateConfig_::colorClause, LongVec_::cont, CRITICAL, damoteConfig, DBevalLSOnSetup(), DBlinkStateEnd(), DBlinkStateInit(), LSPrerouteInfo_::dst, FALSE, LSPrerouteInfo_::id, LSPRequest_::id, longListCopy, longListEnd, longListInit, longListPushBack, LSPRequest_::path, DAMOTEConfig_::predicateConfig, LSPRequest_::rerouteInfo, LSPrerouteInfo_::src, LongVec_::top, and TRUE.

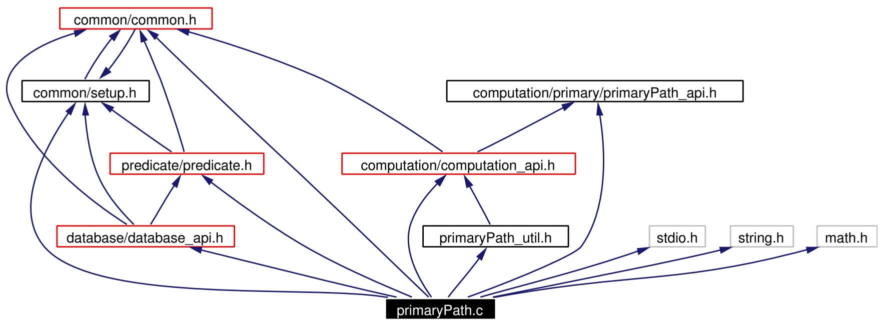
Referenced by fillTopo(), and isValidLSPLink().

```
{
    if (req->id == 1781) {
        int olivier;
        fprintf(stderr,"\nbackup %ld: ", req->id);
        for (olivier=0;olivier<req->path.top;olivier++){
            fprintf(stderr,"%ld ", req->path.cont[olivier]);
        }
    }

    LongList recPath;
    DBLinkState newLS;
    bool gate=TRUE,tmpgate,completePath=TRUE;
    long i;

    if (ls == NULL || req == NULL || gain==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    if (DBlinkStateInit(&newLS)<0)
    {
        addError(CRITICAL,"Unable to initialize link state in %s at line %d",
                 __FILE__,__LINE__);
        return FALSE;
    }

    for (i=0;i<req->path.top && completePath;i++)
    {
        if (req->path.cont[i]<0)
        {
            completePath=FALSE;
        }
    }

    if (! completePath)
    {
        if (longListInit(&recPath,-1)<0)
        {
            addError(CRITICAL,"Unable to initialize path record structure in %s at line %d",
                     __FILE__,__LINE__);
            return FALSE;
        }
        if (longListCopy(&recPath,&(req->path))<0)
        {
            addError(CRITICAL,"Unable to copy path into record structure in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&recPath);
            return FALSE;
        }
        req->path.top=0;
        if (longListPushBack(&(req->path),src)<0 ||
            longListPushBack(&(req->path),dst)<0)
        {
            addError(CRITICAL,"Unable to forge path into request in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&recPath);
            return FALSE;
        }
    }

    if (req->id == 1781) {
        int olivier;
        fprintf(stderr,"\nbackup %ld: ", req->id);
        for (olivier=0;olivier<req->path.top;olivier++){
            fprintf(stderr,"%ld ", req->path.cont[olivier]);
        }
    }


    if (DBevalLSOnSetup(dataBase,src,dst,&newLS,ls,req)<0)
    {
        addError(CRITICAL,"Unable to update link state in %s at line %d",
                 __FILE__,__LINE__);
        if (!completePath)
        {
            longListEnd(&recPath);
        }
        return FALSE;
    }

    if (damoteConfig.predicateConfig.capacityClause)
    {
        tmpgate=capacityClause(&newLS,req,gain);
        gate=gate && tmpgate;
    }

    if (damoteConfig.predicateConfig.colorClause)
    {
        tmpgate=colorClause(&newLS,req);
        gate=gate && tmpgate;
    }

    if (req->rerouteInfo.id >= 0 && req->rerouteInfo.src==src
        && req->rerouteInfo.dst==dst)
    {
        gate= FALSE;
    }

    if (!completePath)
    {
        if (longListCopy(&(req->path),&recPath)<0)
        {
            addError(CRITICAL,"Unable to restore path into request in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&recPath);
            return FALSE;
        }
        longListEnd(&recPath);
    }

    DBlinkStateEnd(&newLS);

    return gate;
}
```

4.23 primaryPath.c File Reference

```
#include "computation/computation_api.h"
#include "database/database_api.h"
#include "common/common.h"
#include "common/setup.h"
#include "predicate/predicate.h"
#include "primaryPath_api.h"
#include "primaryPath_util.h"
#include <stdio.h>
#include <string.h>
#include <math.h>
```

Include dependency graph for primaryPath.c:



Functions

- int computePrimaryPath (DataBase ∗dataBase, LSPRequest ∗req)

    Primary LSP computation function.

- BKConnectVec ∗ bkConnectVecNew (long size)
- int bkConnectVecInit (BKConnectVec ∗vec, long size)
- int bkConnectVecEnd (BKConnectVec ∗vec)
- int bkConnectVecDestroy (BKConnectVec ∗vec)
- int bkConnectVecCopy (BKConnectVec ∗dst, BKConnectVec ∗src)
- int bkConnectVecPushBack (BKConnectVec ∗vec, BKConnect ∗val)
- int bkConnectVecPopBack (BKConnectVec ∗vec, BKConnect ∗val)
- int bkConnectVecResize (BKConnectVec ∗vec, long newsize)
- int bkConnectVecGet (BKConnectVec ∗vec, long index, BKConnect ∗val)
- int bkConnectVecSet (BKConnectVec ∗vec, long index, BKConnect ∗val)
- BKNodeVec ∗ bkNodeVecNew (long size)
- int bkNodeVecInit (BKNodeVec ∗vec, long size)
- int bkNodeVecEnd (BKNodeVec ∗vec)
- int bkNodeVecDestroy (BKNodeVec ∗vec)
- int bkNodeVecPushBack (BKNodeVec ∗vec, BKNode ∗val)

- int bkNodeVecPopBack (BKNodeVec ∗vec, BKNode ∗val)
- int bkNodeVecResize (BKNodeVec ∗vec, long newsize)
- BKNode ∗ bkNodeVecGet (BKNodeVec ∗vec, long index)
- int bkNodeVecSet (BKNodeVec ∗vec, long index, BKNode ∗val)
- int initTopo (BKTopology ∗topo, long size)
- int endTopo (BKTopology ∗topo)
- int fillTopo (DataBase ∗dataBase, LSPRequest ∗req, BKTopology ∗topo)
- int printTopo (BKTopology ∗topo)
- int getRequestSrc (LSPRequest ∗req)
- int getRequestDst (LSPRequest ∗req)
- int updateRequest (BKTopology ∗topo, LSPRequest ∗req)
- int bellmanKalaba (BKTopology ∗topo, LSPRequest ∗req)
- int initScore (long src, BKTopology ∗topo)
- double makeScore (BKTopology ∗topo, LSPRequest ∗req, long src, long dst, BKConnect ∗connect)
- int updateNodeInfoOnElect (BKTopology ∗topo, LSPRequest ∗req, long src, long dst, BKConnect ∗connect)
- int activateNodeInfo (BKTopology ∗topo, long nd)
- int noLoop (BKTopology ∗topo, long src, long dst)

4.23.1 Function Documentation

4.23.1.1 int activateNodeInfo (BKTopology ∗ *topo*, long *nd*)

Definition at line 1562 of file primaryPath.c.

References addError(), BKNodeVec_::cont, LongVec_::cont, CRITICAL, damoteConfig, PrimaryComputationConfig_::loadBal, NB_OA, BKTopology_::nodeInd, BKTopology_::nodeVec, and DAMOTEConfig_::primaryComputationConfig.

Referenced by bellmanKalaba().

```
1563 {
1564     long i;
1565 
1566     if (topo == NULL)
1567     {
1568         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1569                  __FILE__,__LINE__);
1570         return -1;
1571     }
1572 
1573     for (i=0;i<NB_OA;i++)
1574     {
1575         if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
1576         {
1577             topo->nodeVec.cont[topo->nodeInd.cont[nd]].info.sum[i]=
1578                 topo->nodeVec.cont[topo->nodeInd.cont[nd]].info.newSum[i];
1579         }
1580     }
1581 
1582     return 0;
1583 }
```

4.23.1.2 int bellmanKalaba (BKTopology ∗ *topo*, LSPRequest ∗ *req*)

Definition at line 1171 of file primaryPath.c.

References activateNodeInfo(), addError(), bkNodeVecGet(), calloc, BKNodeVec_::cont, LongVec_::cont, BKConnectVec_::cont, BKNodeInfo_::cost, CRITICAL, DIGIT_PRECISION, FALSE, free, getRequestSrc(), BKNode_::info, initScore(), BKNode_::inNeighb, longListEnd, longListInit, longListPushBack, makeScore(), BKConnect_::neighbId, BKNode_::neighbInd, BKNodeInfo_::newCost, BKNodeInfo_::newNeighbInd, BKTopology_::nodeInd, BKTopology_::nodeVec, noLoop(), LongVec_::top, BKConnectVec_::top, TRUE, and updateNodeInfoOnElect().

Referenced by computePrimaryPath().

```
{
    LongList activeNodes;
    BKNode *tmpNode;
    bool done=FALSE;
    int *activeFlags;
    long src,i,j,k,nd,top,threshold,size,iter=0;
    double tmpCost;


    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((src=getRequestSrc(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    size=topo->nodeInd.top;

    if (longListInit(&activeNodes,size)<0)
    {
        addError(CRITICAL,"Unable to initialize the active nodes list in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((activeFlags = (int*) calloc(size,sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&activeNodes);
        return -1;
    }

    if (src>=size)
    {
        addError(CRITICAL,"Inexistent node in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&activeNodes);
        free(activeFlags);
        return -1;
    }
    if (initScore(src,topo)<0)
    {
        addError(CRITICAL,"Unable to initialize scores in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&activeNodes);
        free(activeFlags);
        return -1;
    }
    top=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.top;
    for (i=0;i<top;i++)
    {
        nd=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.cont[i].neighbId;
        if (nd>=size)
        {
            addError(CRITICAL,"Inexistent node in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }

        if (longListPushBack(&activeNodes,nd)<0)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }

        if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }
        for (k=0;(k<tmpNode->inNeighb.top) && (tmpNode->inNeighb.cont[k].neighbId!=src);k++);
        if (k>=tmpNode->inNeighb.top)
        {
            addError(CRITICAL,"Topology unconsistancy in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }
        tmpNode->info.cost=makeScore(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
        tmpNode->info.newCost=tmpNode->info.cost;
        tmpNode->neighbInd=k;
        tmpNode->info.newNeighbInd=tmpNode->neighbInd;
        updateNodeInfoOnElect(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
        activateNodeInfo(topo,nd);
        activeFlags[nd]=1;
    }
    activeFlags[src]=2;

    while (!done)
    {
        iter++;
        done=TRUE;
        threshold=activeNodes.top;
        for (i=0;i<threshold;i++)
        {
            top=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.top;
            for (j=0;j<top;j++)
            {
                nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.cont[j].neigh
                if (nd>=size)
                {
                    addError(CRITICAL,"Inexistent node in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&activeNodes);
                    free(activeFlags);
                    return -1;
                }

                if (activeFlags[nd]!=0 && noLoop(topo,nd,activeNodes.cont[i]))
                {
                    tmpCost=makeScore(topo,req,nd,activeNodes.cont[i],
                                      &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].in
                    if (tmpCost-topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost
                    {
                        done=FALSE;
                        topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost=tmpC
                        topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd
                        updateNodeInfoOnElect(topo,req,nd,activeNodes.cont[i],
                                              &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont
                    }
                }
            }


            if (activeFlags[activeNodes.cont[i]]==1)
            {
                top=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.top;
                for (j=0;j<top;j++)
                {
                    nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.cont[j].
                    if (nd>=size)
                    {
                        addError(CRITICAL,"Inexistent node in %s at line %d",
                                 __FILE__,__LINE__);
                        longListEnd(&activeNodes);
                        free(activeFlags);
                        return -1;
                    }

                    if (activeFlags[nd]==0)
                    {
                        done=FALSE;

                        if (longListPushBack(&activeNodes,nd)<0)
                        {
                            addError(CRITICAL,"Undetermined error in %s at line %d",
                                     __FILE__,__LINE__);
                            longListEnd(&activeNodes);
                            free(activeFlags);
                            return -1;
                        }

                        if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
                        {
                            addError(CRITICAL,"Undetermined error in %s at line %d",
                                     __FILE__,__LINE__);
                            longListEnd(&activeNodes);
                            free(activeFlags);
                            return -1;
                        }
                        for (k=0;(k<tmpNode->inNeighb.top) &&
                                 (tmpNode->inNeighb.cont[k].neighbId!=activeNodes.cont[i]);k++);
                        if (k>=tmpNode->inNeighb.top)
                        {
                            addError(CRITICAL,"Topology unconsistancy in %s at line %d",
                                     __FILE__,__LINE__);
                            longListEnd(&activeNodes);
                            free(activeFlags);
                            return -1;
                        }
```

```
                            tmpNode->info.cost=makeScore(topo,req,activeNodes.cont[i],nd,&tmpNode->inNeig
                            tmpNode->info.newCost=tmpNode->info.cost;
                            tmpNode->neighbInd=k;
                            tmpNode->info.newNeighbInd=tmpNode->neighbInd;
                            updateNodeInfoOnElect(topo,req,activeNodes.cont[i],nd,&tmpNode->inNeighb.cont
                            activateNodeInfo(topo,nd);
                            activeFlags[nd]=1;
                        }
                    }
                    activeFlags[activeNodes.cont[i]]=2;
                }
                else if (activeFlags[activeNodes.cont[i]]==0)
                {
                    addError(CRITICAL,"Internal unconsistancy in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&activeNodes);
                    free(activeFlags);
                    return -1;
                }
            }
            for (i=0;i<threshold;i++)
            {
                if (activeFlags[activeNodes.cont[i]]==2)
                {
                    topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost=
                        topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost;
                    topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].neighbInd=
                        topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd;
                    activateNodeInfo(topo,activeNodes.cont[i]);
                }
            }
        }

        longListEnd(&activeNodes);
        free(activeFlags);

#ifdef DEBUG
        printf("Bellman-Kalaba : %ld iterations \n",iter);
#endif

        return 0;
}
```

4.23.1.3 int bkConnectVecCopy (BKConnectVec ∗ *dst*, BKConnectVec ∗ *src*)

Definition at line 186 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, realloc, BKConnectVec_::size, and BKConnectVec_::top.

Referenced by bkNodeVecPopBack(), bkNodeVecPushBack(), and bkNodeVecSet().

```
{
        BKConnect *ptr=NULL;

        if (dst == NULL || dst->cont == NULL ||
            src == NULL || src->cont == NULL)
        {
            addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        if (dst->size < src->size)
```

```
    {
        if ((ptr=(BKConnect*) realloc(dst->cont,src->size*sizeof(BKConnect)))==NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            dst->cont=ptr;
            dst->size=src->size;
        }
    }

    memcpy(dst->cont,src->cont,src->size*sizeof(BKConnect));
    dst->top=src->top;

    return 0;
}
```

4.23.1.4 int bkConnectVecDestroy (BKConnectVec ∗ *vec*)

Definition at line 171 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, and free.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.23.1.5 int bkConnectVecEnd (BKConnectVec ∗ *vec*)

Definition at line 154 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, free, BKConnectVec_::size, and BKConnect-Vec_::top.

Referenced by bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecInit(), bkNodeVecNew(), and fill-Topo().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.23.1.6 int bkConnectVecGet (BKConnectVec ∗ *vec*, long *index*, BKConnect ∗ *val*)

Definition at line 297 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, BKConnect_::neighbId, and BKConnectVec_::size.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[index].neighbId = val->neighbId;
    vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!

    return 0;
}
```

4.23.1.7 int bkConnectVecInit (BKConnectVec ∗ *vec*, long *size*)

Definition at line 126 of file primaryPath.c.

References addError(), BKCONNECTVEC_INITSIZE, calloc, and CRITICAL.

Referenced by bkNodeVecInit(), bkNodeVecNew(), bkNodeVecResize(), and fillTopo().

```
{
    BKConnect *ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = BKCONNECTVEC_INITSIZE;

    if ((ptr = (BKConnect*) calloc(size,sizeof(BKConnect))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.23.1.8 BKConnectVec∗ bkConnectVecNew (long *size*)

Definition at line 96 of file primaryPath.c.

```
{
    BKConnectVec *vec=NULL;
    BKConnect *ptr=NULL;

    if ((vec = calloc(1,sizeof(BKConnectVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = BKCONNECTVEC_INITSIZE;

    if ((ptr = (BKConnect*) calloc(size,sizeof(BKConnect))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.23.1.9 int bkConnectVecPopBack (BKConnectVec ∗ *vec*, BKConnect ∗ *val*)

Definition at line 250 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, BKConnect_::neighbId, and BKConnectVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    val->neighbId = vec->cont[vec->top - 1].neighbId;
    val->linkState = vec->cont[vec->top--].linkState;

    return 0;
}
```

4.23.1.10 int bkConnectVecPushBack (BKConnectVec ∗ *vec*, BKConnect ∗ *val*)

Definition at line 219 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, BKConnect_::neighbId, realloc, BKConnectVec_::size, and BKConnectVec_::top.

Referenced by fillTopo().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if ((ptr = realloc(vec->cont, vec->size *
                           2 * sizeof(BKConnect))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        vec->size *= 2;
        vec->cont = ptr;
    }

    vec->cont[vec->top].neighbId = val->neighbId;
    vec->cont[vec->top++].linkState = val->linkState; // pointeur directement sur la DB!

    return 0;
}
```

4.23.1.11 int bkConnectVecResize (BKConnectVec ∗ *vec*, long *newsize*)

Definition at line 272 of file primaryPath.c.

References addError(), BKConnect, BKConnectVec_::cont, CRITICAL, realloc, and BKConnectVec_::size.

Referenced by bkConnectVecSet().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = realloc(vec->cont, newsize*sizeof(BKConnect))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont = ptr;
    memset(ptr+ (vec->size * sizeof(BKConnect)), 0, (newsize - vec->size)*sizeof(BKConnect));
    vec->size = newsize;

    return 0;
}
```

4.23.1.12 int bkConnectVecSet (BKConnectVec ∗ *vec*, long *index*, BKConnect ∗ *val*)

Definition at line 326 of file primaryPath.c.

References addError(), bkConnectVecResize(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, max, BKConnect_::neighbId, BKConnectVec_::size, and BKConnectVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        if (bkConnectVecResize(vec,max(vec->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    vec->cont[index].neighbId = val->neighbId;
    vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!
    vec->top=max(vec->top,index+1);

    return 0;
}
```

4.23.1.13 int bkNodeVecDestroy (BKNodeVec ∗ *vec*)

Definition at line 509 of file primaryPath.c.

References addError(), bkConnectVecEnd(), BKNodeVec_::cont, CRITICAL, free, BKNode_::inNeighb, BKNode_::outNeighb, and BKNodeVec_::size.

```
{
    long i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<vec->size;i++)
    {
        bkConnectVecEnd(&vec->cont[i].inNeighb);
        bkConnectVecEnd(&vec->cont[i].outNeighb);
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.23.1.14 int bkNodeVecEnd (BKNodeVec ∗ *vec*)

Definition at line 484 of file primaryPath.c.

References addError(), bkConnectVecEnd(), BKNodeVec_::cont, CRITICAL, free, BKNode_::inNeighb, BKNode_::outNeighb, BKNodeVec_::size, and BKNodeVec_::top.

Referenced by endTopo(), and initTopo().

```
{
    long i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<vec->size;i++)
    {
        bkConnectVecEnd(&vec->cont[i].inNeighb);
        bkConnectVecEnd(&vec->cont[i].outNeighb);
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.23.1.15 BKNode∗ bkNodeVecGet (BKNodeVec ∗ *vec*, long *index*)

Definition at line 640 of file primaryPath.c.

References addError(), BKNodeVec_::cont, CRITICAL, and BKNodeVec_::size.

Referenced by bellmanKalaba(), printTopo(), and updateRequest().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    return vec->cont+index;
}
```

4.23.1.16 int bkNodeVecInit (BKNodeVec ∗ *vec*, long *size*)

Definition at line 426 of file primaryPath.c.

References addError(), bkConnectVecEnd(), bkConnectVecInit(), BKNODEVEC_INITSIZE, calloc, BKNodeVec_::cont, CRITICAL, free, BKNodeVec_::size, and BKNodeVec_::top.

Referenced by initTopo().

```
{
    BKNode* ptr=NULL;
    long i,j;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = BKNODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(BKNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<size;i++)
    {
        if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)
        {
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(ptr);
            return -1;
        }
        else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
        {
            bkConnectVecEnd(&ptr[i].inNeighb);
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(ptr);
            return -1;
        }
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.23.1.17 BKNodeVec∗ bkNodeVecNew (long *size*)

Definition at line 364 of file primaryPath.c.

References addError(), bkConnectVecEnd(), bkConnectVecInit(), BKNODEVEC_INITSIZE, calloc, BKNodeVec_::cont, CRITICAL, free, BKNodeVec_::size, and BKNodeVec_::top.

```
{
    BKNodeVec* vec=NULL;
    BKNode* ptr=NULL;
    long i,j;

    if ((vec = calloc(1,sizeof(BKNodeVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = BKNODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(BKNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    for (i=0;i<size;i++)
    {
        if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)
        {
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(vec);
            free(ptr);
            return NULL;
        }
        else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
        {
            bkConnectVecEnd(&ptr[i].inNeighb);
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(vec);
            free(ptr);
            return NULL;
        }
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.23.1.18 int bkNodeVecPopBack (BKNodeVec ∗ *vec*, BKNode ∗ *val*)

Definition at line 569 of file primaryPath.c.

References addError(), bkConnectVecCopy(), BKNodeVec_::cont, CRITICAL, BKNode_::inNeighb, BKNode_::neighbInd, BKNode_::nodeId, BKNode_::outNeighb, and BKNodeVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (bkConnectVecCopy(&val->inNeighb,&vec->cont[vec->top-1].inNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    if (bkConnectVecCopy(&val->outNeighb,&vec->cont[vec->top-1].outNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    val->nodeId = vec->cont[vec->top-1].nodeId;
    val->neighbInd = vec->cont[vec->top--].neighbInd;

    return 0;
}
```

4.23.1.19 int bkNodeVecPushBack (BKNodeVec ∗ *vec*, BKNode ∗ *val*)

Definition at line 532 of file primaryPath.c.

References addError(), bkConnectVecCopy(), bkNodeVecResize(), BKNodeVec_::cont, CRITICAL, BKNode_::inNeighb, BKNode_::neighbInd, BKNode_::nodeId, BKNode_::outNeighb, BKNodeVec_::size, and BKNodeVec_::top.

Referenced by fillTopo().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if (bkNodeVecResize(vec,vec->size*2)<0)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (bkConnectVecCopy(&vec->cont[vec->top].inNeighb,&val->inNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    if (bkConnectVecCopy(&vec->cont[vec->top].outNeighb,&val->outNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    vec->cont[vec->top].nodeId = val->nodeId;
    vec->cont[vec->top++].neighbInd = val->neighbInd;

    return 0;
}
```

4.23.1.20 int bkNodeVecResize (BKNodeVec ∗ *vec*, long *newsize*)

Definition at line 603 of file primaryPath.c.

References addError(), bkConnectVecInit(), BKNode, BKNodeVec_::cont, CRITICAL, realloc, and BKNodeVec_::size.

Referenced by bkNodeVecPushBack(), and bkNodeVecSet().

```
{
    void *ptr=NULL;
    long i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = (BKNode*) realloc(vec->cont, newsize*sizeof(BKNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(ptr+(vec->size*sizeof(BKNode)) , 0, (newsize-vec->size)*sizeof(BKNode));
    vec->cont = ptr;

    for (i=vec->size;i<newsize;i++)
    {
        if (bkConnectVecInit(&((BKNode*) ptr)[i].inNeighb,-1)<0 ||
            bkConnectVecInit(&((BKNode*) ptr)[i].outNeighb,-1)<0)
        {
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }
    vec->size = newsize;

    return 0;
}
```

4.23.1.21 int bkNodeVecSet (BKNodeVec ∗ *vec*, long *index*, BKNode ∗ *val*)

Definition at line 666 of file primaryPath.c.

References addError(), bkConnectVecCopy(), bkNodeVecResize(), BKNodeVec_::cont, CRITICAL, BKNode_::inNeighb, max, BKNode_::neighbInd, BKNode_::nodeId, BKNode_::outNeighb, BKNodeVec_::size, and BKNodeVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        if (bkNodeVecResize(vec,max(vec->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize node vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (bkConnectVecCopy(&vec->cont[index].inNeighb,&val->inNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    if (bkConnectVecCopy(&vec->cont[index].outNeighb,&val->outNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    vec->cont[index].nodeId = val->nodeId;
    vec->cont[index].neighbInd = val->neighbInd;
    vec->top=max(vec->top,index+1);

    return 0;
}
```

4.23.1.22 int computePrimaryPath (DataBase ∗ *dataBase*, LSPRequest ∗ *req*)

Primary LSP computation function.

Parameters:

dataBase the general database containing topology

req the request containing information about the lsp to be computed

Definition at line 21 of file primaryPath.c.

References addError(), bellmanKalaba(), CRITICAL, endTopo(), fillTopo(), getRequestSrc(), initTopo(), and updateRequest().

```
{
    BKTopology topo;
    long src;

#if defined LINUX && defined TIME4
    struct timezone tz;
    struct timeval  t1,t2;
#endif

    if (dataBase == NULL || req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

#if defined LINUX && defined TIMING && defined TIME4
    gettimeofday(&t1, &tz);
#endif

    if ((src=getRequestSrc(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (initTopo(&topo,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the topology structure in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (fillTopo(dataBase,req,&topo)<0)
    {
        addError(CRITICAL,"Unable to build topology in %s at line %d",
                 __FILE__,__LINE__);
        endTopo(&topo);
        return -1;
    }
    //printTopo(&topo);

    if (bellmanKalaba(&topo,req)<0)
    {
        addError(CRITICAL,"Bellman-Kalaba failure in %s at line %d",
                 __FILE__,__LINE__);
        endTopo(&topo);
        return -1;
    }

    if (updateRequest(&topo,req)<0)
    {
        addError(CRITICAL,"Unable to update requested path in %s at line %d",
                 __FILE__,__LINE__);
        endTopo(&topo);
        return -1;
    }

#if defined LINUX && defined TIMING && defined TIME4
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time for calculation of primary path : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif

    endTopo(&topo);

    return 0;
}
```

4.23.1.23 int endTopo (BKTopology ∗ *topo*)

Definition at line 742 of file primaryPath.c.

References addError(), bkNodeVecEnd(), CRITICAL, longVecEnd(), BKTopology_::nodeInd, and BK-Topology_::nodeVec.

Referenced by computePrimaryPath().

```
{
```

```
744     if (topo == NULL)
745     {
746         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
747                  __FILE__,__LINE__);
748         return -1;
749     }
750 
751     bkNodeVecEnd(&topo->nodeVec);
752     longVecEnd(&topo->nodeInd);
753 
754     return 0;
755 }
```

4.23.1.24 int fillTopo (DataBase ∗ *dataBase*, LSPRequest ∗ *req*, BKTopology ∗ *topo*)

Definition at line 758 of file primaryPath.c.

References addError(), bkConnectVecEnd(), bkConnectVecInit(), bkConnectVecPushBack(), bkNodeVecPushBack(), calloc, LongVec_::cont, BKNodeVec_::cont, BKConnectVec_::cont, CRITICAL, DBgetLinkState(), DBgetMaxNodeID(), DBgetNbLinks(), DBgetNbNodes(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), free, BKConnectInfo_::gain, getRequestSrc(), BKConnect_::info, BKNode_::inNeighb, isValidRequestLink(), BKConnect_::linkState, longListEnd, longListInit, longListPopBack, longListPushBack, longVecSet(), NB_OA, BKTopology_::nbLinks, BKTopology_::nbNodes, BKConnect_::neighbId, BKNode_::neighbInd, BKNode_::nodeId, BKTopology_::nodeInd, BKTopology_::nodeVec, BKNode_::outNeighb, LongVec_::top, BKConnectVec_::top, and BKNodeVec_::top.

Referenced by computePrimaryPath().

```
759 {
760     LongList toDoNodes;
761     int *activeFlags;
762     LongList *tmpNeighb;
763     long i,j,nd,src,size;
764     BKConnect tmpConn;
765     BKNode tmpNode,*nodePtr;
766 
767 
768     if (dataBase == NULL || req==NULL || topo==NULL)
769     {
770         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
771                  __FILE__,__LINE__);
772         return -1;
773     }
774 
775     if ((src=getRequestSrc(req))<0)
776     {
777         addError(CRITICAL,"Unable to get requested source in %s at line %d",
778                  __FILE__,__LINE__);
779         return -1;
780     }
781 
782     size=DBgetMaxNodeID(dataBase)+1;
783 
784     if (longListInit(&toDoNodes,size)<0)
785     {
786         addError(CRITICAL,"Unable to initialize the active nodes list in %s at line %d",
787                  __FILE__,__LINE__);
788         return -1;
789     }
790 
791     if ((activeFlags = (int*) calloc(size,sizeof(long))) == NULL)
792     {
793         addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        return -1;
    }

    memset(&tmpNode,0,sizeof(BKNode));
    if (bkConnectVecInit(&tmpNode.inNeighb,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the temporary node in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        return -1;
    }
    if (bkConnectVecInit(&tmpNode.outNeighb,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the temporary node in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        bkConnectVecEnd(&tmpNode.inNeighb);
        return -1;
    }

    if (longListPushBack(&toDoNodes,src)<0)
    {
        addError(CRITICAL,"Unable to push back on list of longs in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        bkConnectVecEnd(&tmpNode.inNeighb);
        bkConnectVecEnd(&tmpNode.outNeighb);
        return -1;
    }
    activeFlags[src]=1;
    while (toDoNodes.top>0)
    {
        if (longListPopBack(&toDoNodes,&nd)<0)
        {
            addError(CRITICAL,"Unable to pop back on list of longs in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }

        tmpNode.inNeighb.top=0;
        if ((tmpNeighb=DBgetNodeInNeighb(dataBase,nd))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }
        for (i=0;i<tmpNeighb->top;i++)
        {
            if (activeFlags[tmpNeighb->cont[i]]==2)
            {
                nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
                for (j=0;(j<nodePtr->outNeighb.top) && (nodePtr->outNeighb.cont[j].neighbId!=nd);j++);
                if (j<nodePtr->outNeighb.top)
                {
```

```
                tmpConn.neighbId=tmpNeighb->cont[i];
                tmpConn.linkState=nodePtr->outNeighb.cont[j].linkState;
                memset(&tmpConn.info,0,sizeof(BKConnectInfo));
                memcpy(tmpConn.info.gain,nodePtr->outNeighb.cont[j].info.gain,NB_OA*sizeof(double
                if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)
                {
                    addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&toDoNodes);
                    free(activeFlags);
                    bkConnectVecEnd(&tmpNode.inNeighb);
                    bkConnectVecEnd(&tmpNode.outNeighb);
                    return -1;
                }
            }
        }
        else
        {
            tmpConn.neighbId=tmpNeighb->cont[i];
            tmpConn.linkState=DBgetLinkState(dataBase,tmpNeighb->cont[i],nd);
            memset(&tmpConn.info,0,sizeof(BKConnectInfo));
            if (isValidRequestLink(dataBase,tmpNeighb->cont[i],nd,
                                   tmpConn.linkState,req,tmpConn.info.gain))
            {
                if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)
                {
                    addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&toDoNodes);
                    free(activeFlags);
                    bkConnectVecEnd(&tmpNode.inNeighb);
                    bkConnectVecEnd(&tmpNode.outNeighb);
                    return -1;
                }
            }
        }
    }

    tmpNode.outNeighb.top=0;
    if ((tmpNeighb=DBgetNodeOutNeighb(dataBase,nd))==NULL)
    {
        addError(CRITICAL,"Unable to get the list of neighbours in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        bkConnectVecEnd(&tmpNode.inNeighb);
        bkConnectVecEnd(&tmpNode.outNeighb);
        return -1;
    }
    for (i=0;i<tmpNeighb->top;i++)
    {
        if (activeFlags[tmpNeighb->cont[i]]==2)
        {
            nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
            for (j=0;(j<nodePtr->inNeighb.top) && (nodePtr->inNeighb.cont[j].neighbId!=nd);j++);
            if (j<nodePtr->inNeighb.top)
            {
                tmpConn.neighbId=tmpNeighb->cont[i];
                tmpConn.linkState=nodePtr->inNeighb.cont[j].linkState;
                memset(&tmpConn.info,0,sizeof(BKConnectInfo));
                memcpy(tmpConn.info.gain,nodePtr->inNeighb.cont[j].info.gain,NB_OA*sizeof(double))
                if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
                {
                    addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&toDoNodes);
                    free(activeFlags);
```

```
                        bkConnectVecEnd(&tmpNode.inNeighb);
                        bkConnectVecEnd(&tmpNode.outNeighb);
                        return -1;
                    }
                }
            }
            else
            {
                tmpConn.neighbId=tmpNeighb->cont[i];
                tmpConn.linkState=DBgetLinkState(dataBase,nd,tmpNeighb->cont[i]);
                memset(&tmpConn.info,0,sizeof(BKConnectInfo));
                if (isValidRequestLink(dataBase,nd,tmpNeighb->cont[i],
                                       tmpConn.linkState,req,tmpConn.info.gain))
                {
                    if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
                    {
                        addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                                 __FILE__,__LINE__);
                        longListEnd(&toDoNodes);
                        free(activeFlags);
                        bkConnectVecEnd(&tmpNode.inNeighb);
                        bkConnectVecEnd(&tmpNode.outNeighb);
                        return -1;
                    }
                }
            }
            if (activeFlags[tmpNeighb->cont[i]]==0)
            {
                if (longListPushBack(&toDoNodes,tmpNeighb->cont[i])<0)
                {
                    addError(CRITICAL,"Unable to push back on list of longs in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&toDoNodes);
                    free(activeFlags);
                    return -1;
                }
                activeFlags[tmpNeighb->cont[i]]=1;
            }
        }

        tmpNode.nodeId=nd;
        tmpNode.neighbInd=-1;
        if (bkNodeVecPushBack(&topo->nodeVec,&tmpNode)<0)
        {
            addError(CRITICAL,"Unable to push back node in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }

        if (longVecSet(&topo->nodeInd,nd,(topo->nodeVec.top-1))<0)
        {
            addError(CRITICAL,"Unable to set node index in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }

        activeFlags[nd]=2;
    }

    if (((topo->nbNodes=DBgetNbNodes(dataBase))<0)||
        ((topo->nbLinks=DBgetNbLinks(dataBase))<0))
    {
        addError(CRITICAL,"Unable to get number of nodes and links in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        bkConnectVecEnd(&tmpNode.inNeighb);
        bkConnectVecEnd(&tmpNode.outNeighb);
        return -1;
    }

    longListEnd(&toDoNodes);
    free(activeFlags);
    bkConnectVecEnd(&tmpNode.inNeighb);
    bkConnectVecEnd(&tmpNode.outNeighb);

    return 0;
}
```

4.23.1.25 int getRequestDst (LSPRequest ∗ *req*)

Definition at line 1077 of file primaryPath.c.

References addError(), LongVec_::cont, CRITICAL, LSPRequest_::path, and LongVec_::top.

Referenced by updateRequest().

```
{
    if (req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (req->path.top<2 || req->path.cont[0]<0 ||
        req->path.cont[req->path.top-1]<0)
    {
        addError(CRITICAL,"Bad requested path format in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return req->path.cont[req->path.top-1];
}
```

4.23.1.26 int getRequestSrc (LSPRequest ∗ *req*)

Definition at line 1057 of file primaryPath.c.

References addError(), LongVec_::cont, CRITICAL, LSPRequest_::path, and LongVec_::top.

Referenced by bellmanKalaba(), computePrimaryPath(), fillTopo(), and updateRequest().

```
{
    if (req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (req->path.top<2 || req->path.cont[0]<0 ||
        req->path.cont[req->path.top-1]<0)
    {
        addError(CRITICAL,"Bad requested path format in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return req->path.cont[0];
}
```

4.23.1.27 int initScore (long *src*, BKTopology ∗ *topo*)

Definition at line 1402 of file primaryPath.c.

References addError(), DBLinkState_::cap, BKNodeVec_::cont, LongVec_::cont, BKConnectVec_::cont, CRITICAL, damoteConfig, FALSE, BKNode_::inNeighb, BKConnect_::linkState, PrimaryComputationConfig_::loadBal, NB_OA, NB_PREEMPTION, BKTopology_::nodeInd, BKTopology_::nodeVec, DBLinkState_::pbw, DAMOTEConfig_::primaryComputationConfig, BKNodeVec_::top, BKConnectVec_::top, and TRUE.

Referenced by bellmanKalaba().

```
{
    bool process=FALSE;
    long i,j,k,l,top;
    double tmpSum;

    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
    {
        if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
        {
            process=TRUE;
        }
    }

    if (process)
    {
        for (k=0;k<NB_OA;k++)
        {
            topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]=0;
        }
        for (i=0;i<topo->nodeVec.top;i++)
        {
            top=topo->nodeVec.cont[i].inNeighb.top;
            for (j=0;j<top;j++)
            {
                for (k=0;k<NB_OA;k++)
                {
                    tmpSum=0;
                    for (l=0;l<NB_PREEMPTION;l++)
                    {
                        tmpSum+=topo->nodeVec.cont[i].inNeighb.cont[j].linkState->pbw[k][l];
                    }
                    topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]+=tmpSum/
                        topo->nodeVec.cont[i].inNeighb.cont[j].linkState->cap[k];
                }
            }
        }
    }

    return 0;
}
```

4.23.1.28 int initTopo (BKTopology ∗ *topo*, long *size*)

Definition at line 715 of file primaryPath.c.

References addError(), bkNodeVecEnd(), bkNodeVecInit(), CRITICAL, longVecInit(), BKTopology_-::nodeInd, and BKTopology_::nodeVec.

Referenced by computePrimaryPath().

```
{
    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (bkNodeVecInit(&topo->nodeVec,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize node vector in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecInit(&topo->nodeInd,size)<0)
    {
        addError(CRITICAL,"Unable to initialize long vector in %s at line %d",
                 __FILE__,__LINE__);
        bkNodeVecEnd(&topo->nodeVec);
        return -1;
    }

    return 0;
}
```

4.23.1.29 double makeScore (BKTopology ∗ *topo*, LSPRequest ∗ *req*, long *src*, long *dst*, BKConnect ∗ *connect*)

Definition at line 1452 of file primaryPath.c.

References addError(), LSPRequest_::bw, DBLinkState_::cap, BKNodeVec_::cont, LongVec_::cont, CRITICAL, damoteConfig, PrimaryComputationConfig_::delay, BKConnectInfo_::gain, BKConnect_-::info, BKConnect_::linkState, PrimaryComputationConfig_::load, PrimaryComputationConfig_::loadBal, makeRerouteScore(), NB_OA, NB_PREEMPTION, BKTopology_::nbLinks, BKTopology_::nodeInd, BKTopology_::nodeVec, DBLinkState_::pbw, DAMOTEConfig_::primaryComputationConfig, Primary-ComputationConfig_::relLoad, PrimaryComputationConfig_::rerouting, PrimaryComputationConfig_::sq-Load, and PrimaryComputationConfig_::sqRelLoad.

Referenced by bellmanKalaba().

```
{
    double score=0,totBW[NB_OA],newSum,rerouteScore=0;
    long i,j;

    if (topo == NULL || connect == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return HUGE_VAL;
    }

    score=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.cost;

    for (i=0;i<NB_OA;i++)
    {
        totBW[i]=0;
        for (j=0;j<NB_PREEMPTION;j++)
        {
            totBW[i]+=connect->linkState->pbw[i][j];
        }

        if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *((totBW[i]+req->bw[i])/connect->linkState->cap[i])
                *((totBW[i]+req->bw[i])/connect->linkState->cap[i]);
            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *(totBW[i]/connect->linkState->cap[i])
                *(totBW[i]/connect->linkState->cap[i]);

            newSum=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkS
            if (__isinf(newSum))
            {
                return HUGE_VAL;
            }

            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *(-1/(double)topo->nbLinks)*newSum*newSum;
            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *(1/(double)topo->nbLinks)*topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]
                *topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i];
        }
        if (damoteConfig.primaryComputationConfig.load[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.load[i]*req->bw[i];
        }
        if (damoteConfig.primaryComputationConfig.sqLoad[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.sqLoad[i]
                *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i]);
        }
        if (damoteConfig.primaryComputationConfig.relLoad[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.relLoad[i]
                *req->bw[i]/connect->linkState->cap[i];
        }
        if (damoteConfig.primaryComputationConfig.sqRelLoad[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.sqRelLoad[i]
                *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i])
                /(connect->linkState->cap[i]*connect->linkState->cap[i]);
        }
        if (damoteConfig.primaryComputationConfig.delay[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.delay[i]
                *((1/(connect->linkState->cap[i]-totBW[i]-req->bw[i]))
                  -(1/(connect->linkState->cap[i]-totBW[i])));
        }
    }

    for (i=0;i<NB_OA;i++)
    {
        if (damoteConfig.primaryComputationConfig.rerouting[i]!=0)
        {
            rerouteScore+=damoteConfig.primaryComputationConfig.rerouting[i]*
                makeRerouteScore(req,connect->info.gain,connect->linkState,i);
        }
    }

    score+=rerouteScore*(score>0?1:0)*score;

    return score;
}
```

4.23.1.30 int noLoop (BKTopology ∗ *topo*, long *src*, long *dst*)

Definition at line 1586 of file primaryPath.c.

References BKNodeVec_::cont, LongVec_::cont, BKConnectVec_::cont, BKNode_::info, BKNode_::inNeighb, BKNode_::neighbInd, BKNodeInfo_::newNeighbInd, BKNode_::nodeId, BKTopology_::nodeInd, and BKTopology_::nodeVec.

Referenced by bellmanKalaba().

```
{
    BKNode* tmpNode;


    tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[src]];
    while (tmpNode->neighbInd!=-1 && tmpNode->nodeId!=dst)
    {
        tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[tmpNode->inNeighb.cont[tmpNode->info.newNeighb
    }

    if (tmpNode->nodeId==dst)
        return 0;

    return 1;
}
```

4.23.1.31 int printTopo (BKTopology ∗ *topo*)

Definition at line 1015 of file primaryPath.c.

References addError(), bkNodeVecGet(), LongVec_::cont, BKConnectVec_::cont, CRITICAL, BKNode_::inNeighb, BKConnect_::neighbId, BKNode_::neighbInd, BKNode_::nodeId, BKTopology_::nodeInd, BKTopology_::nodeVec, BKNode_::outNeighb, LongVec_::top, and BKConnectVec_::top.

```
{
    BKNode *tmpNode;
    long i,j;

    for (i=0;i<topo->nodeInd.top;i++)
    {
        tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[i]);
        if (tmpNode!=NULL)
        {
            if (i!=tmpNode->nodeId)
            {
                addError(CRITICAL,"Topology unconsistancy in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }

            printf("Node %ld\n--------\n",i);
            printf("Incoming neighboors : \n");

            for (j=0; j<tmpNode->inNeighb.top; j++)
            {
                printf("%ld ", tmpNode->inNeighb.cont[j].neighbId);
            }

            printf("\nOutgoing neighboors : \n");

            for (j=0; j<tmpNode->outNeighb.top; j++)
            {
                printf("%ld ", tmpNode->outNeighb.cont[j].neighbId);
            }
            printf("\n");

            printf("Chosen Neighbour Index: %ld \n",tmpNode->neighbInd);

            printf("\n");
        }
    }

    return 0;
}
```

4.23.1.32 int updateNodeInfoOnElect (BKTopology ∗ *topo*, LSPRequest ∗ *req*, long *src*, long *dst*, BKConnect ∗ *connect*)

Definition at line 1538 of file primaryPath.c.

References addError(), LSPRequest_::bw, DBLinkState_::cap, BKNodeVec_::cont, LongVec_::cont, CRITICAL, damoteConfig, BKConnect_::linkState, PrimaryComputationConfig_::loadBal, NB_OA, BKTopology_::nodeInd, BKTopology_::nodeVec, and DAMOTEConfig_::primaryComputationConfig.

Referenced by bellmanKalaba().

```
{
    long i;

    if (topo == NULL || connect == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
    {
        if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
        {
            topo->nodeVec.cont[topo->nodeInd.cont[dst]].info.newSum[i]=
                topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkStat
        }
    }

    return 0;
}
```

4.23.1.33 int updateRequest (BKTopology ∗ *topo*, LSPRequest ∗ *req*)

Definition at line 1097 of file primaryPath.c.

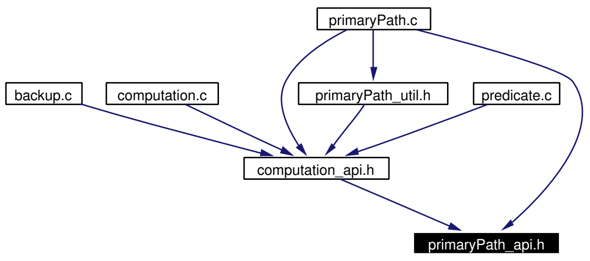
References addError(), bkNodeVecGet(), LongVec_::cont, BKConnectVec_::cont, CRITICAL, getRequestDst(), getRequestSrc(), BKNode_::inNeighb, longListPushBack, BKNode_::neighbInd, BKTopology_::nodeInd, BKTopology_::nodeVec, LSPRequest_::path, and LongVec_::top.

Referenced by computePrimaryPath().

```
{
    BKNode *tmpNode;
    long i,src,dst,nd;

    if (topo == NULL || req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((src=getRequestSrc(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((dst=getRequestDst(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    req->path.top=0;
    nd=dst;
    if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
    {
        addError(CRITICAL,"Undetermined error in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    while (nd!=src)
    {
        if (tmpNode->neighbInd < 0)
        {
            addError(CRITICAL,"Destination unreachable in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        if (longListPushBack(&req->path,nd)<0)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        nd=tmpNode->inNeighb.cont[tmpNode->neighbInd].neighbId;
        if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }
    if (longListPushBack(&req->path,nd)<0)
    {
        addError(CRITICAL,"Undetermined error in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<req->path.top/2;i++)
    {
        nd=req->path.cont[i];
        req->path.cont[i]=req->path.cont[req->path.top-1-i];
        req->path.cont[req->path.top-1-i]=nd;
    }

    return 0;
}
```

4.24 primaryPath_api.h File Reference

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



Functions

- int computePrimaryPath (DataBase ∗, LSPRequest ∗)

  Primary LSP computation function.

4.24.1 Function Documentation

4.24.1.1 int computePrimaryPath (DataBase ∗ *dataBase*, LSPRequest ∗ *req*)

Primary LSP computation function.

Parameters:

dataBase the general database containing topology

req the request containing information about the lsp to be computed

Definition at line 21 of file primaryPath.c.

References addError(), bellmanKalaba(), CRITICAL, endTopo(), fillTopo(), getRequestSrc(), initTopo(), and updateRequest().

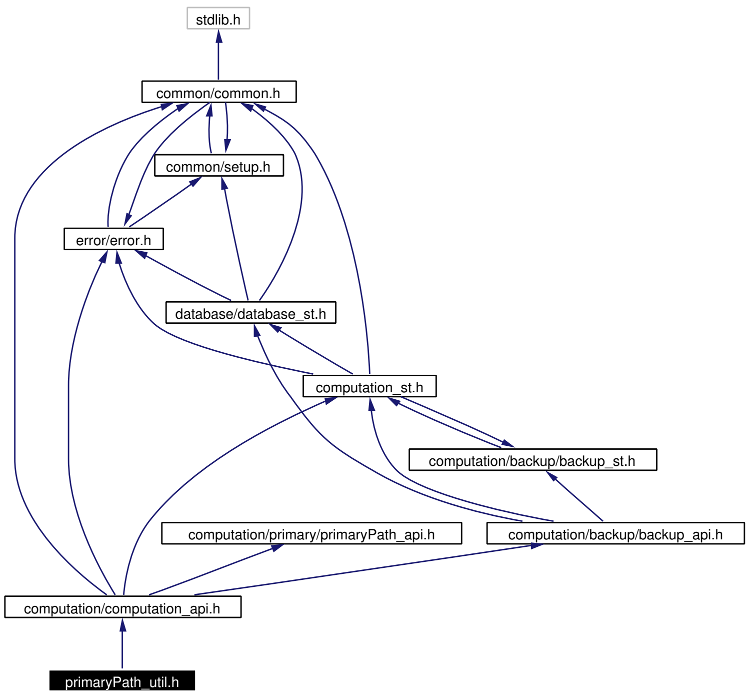
```
{
    BKTopology topo;
    long src;

#if defined LINUX && defined TIME4
    struct timezone tz;
    struct timeval  t1,t2;
#endif

    if (dataBase == NULL || req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

#if defined LINUX && defined TIMING && defined TIME4
    gettimeofday(&t1, &tz);
#endif

```

```
    if ((src=getRequestSrc(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (initTopo(&topo,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the topology structure in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (fillTopo(dataBase,req,&topo)<0)
    {
        addError(CRITICAL,"Unable to build topology in %s at line %d",
                 __FILE__,__LINE__);
        endTopo(&topo);
        return -1;
    }
    //printTopo(&topo);

    if (bellmanKalaba(&topo,req)<0)
    {
        addError(CRITICAL,"Bellman-Kalaba failure in %s at line %d",
                 __FILE__,__LINE__);
        endTopo(&topo);
        return -1;
    }

    if (updateRequest(&topo,req)<0)
    {
        addError(CRITICAL,"Unable to update requested path in %s at line %d",
                 __FILE__,__LINE__);
        endTopo(&topo);
        return -1;
    }

#if defined LINUX && defined TIMING && defined TIME4
    gettimeofday(&t2, &tz);
    fprintf(stderr, "Time for calculation of primary path : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
            (t2.tv_usec - t1.tv_usec) / 1000.0);
#endif

    endTopo(&topo);

    return 0;
}
```

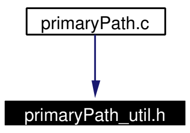
4.25 primaryPath_util.h File Reference

```
#include "computation/computation_api.h"
```

Include dependency graph for primaryPath_util.h:


stdlib.h
common/common.h
common/setup.h
error/error.h
database/database_st.h
computation_st.h
computation/backup/backup_st.h
computation/primary/primaryPath_api.h
computation/backup/backup_api.h
computation/computation_api.h
primaryPath_util.h

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



Data Structures

- struct BKConnect_
- struct BKConnectInfo_
- struct BKConnectVec_
- struct BKNode_
- struct BKNodeInfo_
- struct BKNodeVec_
- struct BKTopology_

Typedefs

- typedef BKConnectInfo_ BKConnectInfo

- typedef BKConnect_ BKConnect
- typedef BKConnectVec_ BKConnectVec
- typedef BKNodeInfo_ BKNodeInfo
- typedef BKNode_ BKNode
- typedef BKNodeVec_ BKNodeVec
- typedef BKTopology_ BKTopology

Functions

- BKConnectVec ∗ bkConnectVecNew (long)
- int bkConnectVecInit (BKConnectVec ∗, long)
- int bkConnectVecEnd (BKConnectVec ∗)
- int bkConnectVecDestroy (BKConnectVec ∗)
- int bkConnectVecCopy (BKConnectVec ∗, BKConnectVec ∗)
- int bkConnectVecPushBack (BKConnectVec ∗, BKConnect ∗)
- int bkConnectVecPopBack (BKConnectVec ∗, BKConnect ∗)
- int bkConnectVecResize (BKConnectVec ∗, long)
- int bkConnectVecGet (BKConnectVec ∗, long, BKConnect ∗)
- int bkConnectVecSet (BKConnectVec ∗, long, BKConnect ∗)
- BKNodeVec ∗ bkNodeVecNew (long)
- int bkNodeVecInit (BKNodeVec ∗, long)
- int bkNodeVecEnd (BKNodeVec ∗)
- int bkNodeVecDestroy (BKNodeVec ∗)
- int bkNodeVecPushBack (BKNodeVec ∗, BKNode ∗)
- int bkNodeVecPopBack (BKNodeVec ∗, BKNode ∗)
- int bkNodeVecResize (BKNodeVec ∗, long)
- BKNode ∗ bkNodeVecGet (BKNodeVec ∗, long)
- int bkNodeVecSet (BKNodeVec ∗, long, BKNode ∗)
- int initTopo (BKTopology ∗, long)
- int endTopo (BKTopology ∗)
- int fillTopo (DataBase ∗, LSPRequest ∗, BKTopology ∗)
- int printTopo (BKTopology ∗)
- int getRequestSrc (LSPRequest ∗)
- int getRequestDst (LSPRequest ∗)
- int updateRequest (BKTopology ∗, LSPRequest ∗)
- int bellmanKalaba (BKTopology ∗, LSPRequest ∗)
- int initScore (long, BKTopology ∗)
- double makeScore (BKTopology ∗, LSPRequest ∗, long, long, BKConnect ∗)
- int updateNodeInfoOnElect (BKTopology ∗, LSPRequest ∗, long, long, BKConnect ∗)
- int activateNodeInfo (BKTopology ∗, long)
- int noLoop (BKTopology ∗, long, long)

4.25.1 Typedef Documentation

4.25.1.1 typedef struct BKConnect_ BKConnect

Referenced by bkConnectVecResize().

4.25.1.2 typedef struct BKConnectInfo_ BKConnectInfo

4.25.1.3 typedef struct BKConnectVec_ BKConnectVec

4.25.1.4 typedef struct BKNode_ BKNode

Referenced by bkNodeVecResize().

4.25.1.5 typedef struct BKNodeInfo_ BKNodeInfo

4.25.1.6 typedef struct BKNodeVec_ BKNodeVec

4.25.1.7 typedef struct BKTopology_ BKTopology

4.25.2 Function Documentation

4.25.2.1 int activateNodeInfo (BKTopology ∗, long)

Definition at line 1562 of file primaryPath.c.

References addError(), LongVec_::cont, BKNodeVec_::cont, CRITICAL, damoteConfig, PrimaryComputationConfig_::loadBal, NB_OA, BKTopology_::nodeInd, BKTopology_::nodeVec, and DAMOTEConfig_::primaryComputationConfig.

Referenced by bellmanKalaba().

```
{
    long i;

    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
    {
        if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
        {
            topo->nodeVec.cont[topo->nodeInd.cont[nd]].info.sum[i]=
                topo->nodeVec.cont[topo->nodeInd.cont[nd]].info.newSum[i];
        }
    }

    return 0;
}
```

4.25.2.2 int bellmanKalaba (BKTopology ∗, LSPRequest ∗)

Definition at line 1171 of file primaryPath.c.

References activateNodeInfo(), addError(), bkNodeVecGet(), calloc, BKConnectVec_::cont, LongVec_::cont, BKNodeVec_::cont, BKNodeInfo_::cost, CRITICAL, DIGIT_PRECISION, FALSE, free, getRequestSrc(), BKNode_::info, initScore(), BKNode_::inNeighb, longListEnd, longListInit, longListPushBack, makeScore(), BKConnect_::neighbId, BKNode_::neighbInd, BKNodeInfo_::newCost, BKNode-

Info_::newNeighbInd, BKTopology_::nodeInd, BKTopology_::nodeVec, noLoop(), BKConnectVec_::top, LongVec_::top, TRUE, and updateNodeInfoOnElect().

Referenced by computePrimaryPath().

```
{
    LongList activeNodes;
    BKNode *tmpNode;
    bool done=FALSE;
    int *activeFlags;
    long src,i,j,k,nd,top,threshold,size,iter=0;
    double tmpCost;


    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((src=getRequestSrc(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    size=topo->nodeInd.top;

    if (longListInit(&activeNodes,size)<0)
    {
        addError(CRITICAL,"Unable to initialize the active nodes list in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((activeFlags = (int*) calloc(size,sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&activeNodes);
        return -1;
    }

    if (src>=size)
    {
        addError(CRITICAL,"Inexistent node in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&activeNodes);
        free(activeFlags);
        return -1;
    }
    if (initScore(src,topo)<0)
    {
        addError(CRITICAL,"Unable to initialize scores in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&activeNodes);
        free(activeFlags);
        return -1;
    }
    top=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.top;
    for (i=0;i<top;i++)
    {
        nd=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.cont[i].neighbId;
        if (nd>=size)
        {
            addError(CRITICAL,"Inexistent node in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }

        if (longListPushBack(&activeNodes,nd)<0)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }

        if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }
        for (k=0;(k<tmpNode->inNeighb.top) && (tmpNode->inNeighb.cont[k].neighbId!=src);k++);
        if (k>=tmpNode->inNeighb.top)
        {
            addError(CRITICAL,"Topology unconsistancy in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&activeNodes);
            free(activeFlags);
            return -1;
        }
        tmpNode->info.cost=makeScore(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
        tmpNode->info.newCost=tmpNode->info.cost;
        tmpNode->neighbInd=k;
        tmpNode->info.newNeighbInd=tmpNode->neighbInd;
        updateNodeInfoOnElect(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
        activateNodeInfo(topo,nd);
        activeFlags[nd]=1;
    }
    activeFlags[src]=2;

    while (!done)
    {
        iter++;
        done=TRUE;
        threshold=activeNodes.top;
        for (i=0;i<threshold;i++)
        {
            top=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.top;
            for (j=0;j<top;j++)
            {
                nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.cont[j].neigh
                if (nd>=size)
                {
                    addError(CRITICAL,"Inexistent node in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&activeNodes);
                    free(activeFlags);
                    return -1;
                }

                if (activeFlags[nd]!=0 && noLoop(topo,nd,activeNodes.cont[i]))
                {
                    tmpCost=makeScore(topo,req,nd,activeNodes.cont[i],
                                      &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].ir
                    if (tmpCost-topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost
                    {
                        done=FALSE;
                        topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost=tmpC
                        topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd
                        updateNodeInfoOnElect(topo,req,nd,activeNodes.cont[i],
                                              &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont
                    }
                }
            }


            if (activeFlags[activeNodes.cont[i]]==1)
            {
                top=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.top;
                for (j=0;j<top;j++)
                {
                    nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.cont[j].
                    if (nd>=size)
                    {
                        addError(CRITICAL,"Inexistent node in %s at line %d",
                                 __FILE__,__LINE__);
                        longListEnd(&activeNodes);
                        free(activeFlags);
                        return -1;
                    }

                    if (activeFlags[nd]==0)
                    {
                        done=FALSE;

                        if (longListPushBack(&activeNodes,nd)<0)
                        {
                            addError(CRITICAL,"Undetermined error in %s at line %d",
                                     __FILE__,__LINE__);
                            longListEnd(&activeNodes);
                            free(activeFlags);
                            return -1;
                        }

                        if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
                        {
                            addError(CRITICAL,"Undetermined error in %s at line %d",
                                     __FILE__,__LINE__);
                            longListEnd(&activeNodes);
                            free(activeFlags);
                            return -1;
                        }
                        for (k=0;(k<tmpNode->inNeighb.top) &&
                                 (tmpNode->inNeighb.cont[k].neighbId!=activeNodes.cont[i]);k++);
                        if (k>=tmpNode->inNeighb.top)
                        {
                            addError(CRITICAL,"Topology unconsistancy in %s at line %d",
                                     __FILE__,__LINE__);
                            longListEnd(&activeNodes);
                            free(activeFlags);
                            return -1;
                        }
                        tmpNode->info.cost=makeScore(topo,req,activeNodes.cont[i],nd,&tmpNode->inNeig
                        tmpNode->info.newCost=tmpNode->info.cost;
                        tmpNode->neighbInd=k;
                        tmpNode->info.newNeighbInd=tmpNode->neighbInd;
                        updateNodeInfoOnElect(topo,req,activeNodes.cont[i],nd,&tmpNode->inNeighb.cont
                        activateNodeInfo(topo,nd);
                        activeFlags[nd]=1;
                    }
                }
                activeFlags[activeNodes.cont[i]]=2;
            }
            else if (activeFlags[activeNodes.cont[i]]==0)
            {
                addError(CRITICAL,"Internal unconsistancy in %s at line %d",
                         __FILE__,__LINE__);
                longListEnd(&activeNodes);
                free(activeFlags);
                return -1;
            }
        }
        for (i=0;i<threshold;i++)
        {
            if (activeFlags[activeNodes.cont[i]]==2)
            {
                topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost=
                    topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost;
                topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].neighbInd=
                    topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd;
                activateNodeInfo(topo,activeNodes.cont[i]);
            }
        }
    }

    longListEnd(&activeNodes);
    free(activeFlags);

#ifdef DEBUG
    printf("Bellman-Kalaba : %ld iterations \n",iter);
#endif

    return 0;
}
```

4.25.2.3 int bkConnectVecCopy (BKConnectVec *, BKConnectVec *)

Definition at line 186 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, realloc, BKConnectVec_::size, and BKConnectVec_::top.

Referenced by bkNodeVecPopBack(), bkNodeVecPushBack(), and bkNodeVecSet().

```
{
    BKConnect *ptr=NULL;

    if (dst == NULL || dst->cont == NULL ||
        src == NULL || src->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (dst->size < src->size)
    {
        if ((ptr=(BKConnect*) realloc(dst->cont,src->size*sizeof(BKConnect)))==NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        else
        {
            dst->cont=ptr;
            dst->size=src->size;
        }
    }

    memcpy(dst->cont,src->cont,src->size*sizeof(BKConnect));
    dst->top=src->top;

    return 0;
}
```

4.25.2.4 int bkConnectVecDestroy (BKConnectVec ∗)

Definition at line 171 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, and free.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.25.2.5 int bkConnectVecEnd (BKConnectVec ∗)

Definition at line 154 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, free, BKConnectVec_::size, and BKConnect-Vec_::top.

Referenced by bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecInit(), bkNodeVecNew(), and fill-Topo().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.25.2.6 int bkConnectVecGet (BKConnectVec ∗, long, BKConnect ∗)

Definition at line 297 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, BKConnect_::neighbId, and BKConnectVec_::size.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont[index].neighbId = val->neighbId;
    vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!

    return 0;
}
```

4.25.2.7 int bkConnectVecInit (BKConnectVec ∗, long)

Definition at line 126 of file primaryPath.c.

References addError(), BKCONNECTVEC_INITSIZE, calloc, and CRITICAL.

Referenced by bkNodeVecInit(), bkNodeVecNew(), bkNodeVecResize(), and fillTopo().

```
{
    BKConnect *ptr=NULL;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = BKCONNECTVEC_INITSIZE;

    if ((ptr = (BKConnect*) calloc(size,sizeof(BKConnect))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.25.2.8 BKConnectVec∗ bkConnectVecNew (long)

Definition at line 96 of file primaryPath.c.

```
{
   BKConnectVec *vec=NULL;
   BKConnect *ptr=NULL;

    if ((vec = calloc(1,sizeof(BKConnectVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = BKCONNECTVEC_INITSIZE;

    if ((ptr = (BKConnect*) calloc(size,sizeof(BKConnect))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.25.2.9 int bkConnectVecPopBack (BKConnectVec ∗, BKConnect ∗)

Definition at line 250 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, BKConnect_::neighbId, and BKConnectVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    val->neighbId = vec->cont[vec->top - 1].neighbId;
    val->linkState = vec->cont[vec->top--].linkState;

    return 0;
}
```

4.25.2.10 int bkConnectVecPushBack (BKConnectVec *, BKConnect *)

Definition at line 219 of file primaryPath.c.

References addError(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, BKConnect_::neighbId, realloc, BKConnectVec_::size, and BKConnectVec_::top.

Referenced by fillTopo().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if ((ptr = realloc(vec->cont, vec->size *
                           2 * sizeof(BKConnect))) == NULL)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        vec->size *= 2;
        vec->cont = ptr;
    }

    vec->cont[vec->top].neighbId = val->neighbId;
    vec->cont[vec->top++].linkState = val->linkState; // pointeur directement sur la DB!

    return 0;
}
```

4.25.2.11 int bkConnectVecResize (BKConnectVec *, long)

Definition at line 272 of file primaryPath.c.

References addError(), BKConnect, BKConnectVec_::cont, CRITICAL, realloc, and BKConnectVec_::size.

Referenced by bkConnectVecSet().

```
{
    void* ptr=NULL;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = realloc(vec->cont, newsize*sizeof(BKConnect))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    vec->cont = ptr;
    memset(ptr+ (vec->size * sizeof(BKConnect)), 0, (newsize - vec->size)*sizeof(BKConnect));
    vec->size = newsize;

    return 0;
}
```

4.25.2.12 int bkConnectVecSet (BKConnectVec ∗, long, BKConnect ∗)

Definition at line 326 of file primaryPath.c.

References addError(), bkConnectVecResize(), BKConnectVec_::cont, CRITICAL, BKConnect_::linkState, max, BKConnect_::neighbId, BKConnectVec_::size, and BKConnectVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        if (bkConnectVecResize(vec,max(vec->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    vec->cont[index].neighbId = val->neighbId;
    vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!
    vec->top=max(vec->top,index+1);

    return 0;
}
```

4.25.2.13 int bkNodeVecDestroy (BKNodeVec ∗)

Definition at line 509 of file primaryPath.c.

References addError(), bkConnectVecEnd(), BKNodeVec_::cont, CRITICAL, free, BKNode_::inNeighb, BKNode_::outNeighb, and BKNodeVec_::size.

```
{
    long i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<vec->size;i++)
    {
        bkConnectVecEnd(&vec->cont[i].inNeighb);
        bkConnectVecEnd(&vec->cont[i].outNeighb);
    }

    free(vec->cont);
    free(vec);

    return 0;
}
```

4.25.2.14 int bkNodeVecEnd (BKNodeVec ∗)

Definition at line 484 of file primaryPath.c.

References addError(), bkConnectVecEnd(), BKNodeVec_::cont, CRITICAL, free, BKNode_::inNeighb, BKNode_::outNeighb, BKNodeVec_::size, and BKNodeVec_::top.

Referenced by endTopo(), and initTopo().

```
{
    long i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<vec->size;i++)
    {
        bkConnectVecEnd(&vec->cont[i].inNeighb);
        bkConnectVecEnd(&vec->cont[i].outNeighb);
    }

    free(vec->cont);
    vec->cont = NULL;
    vec->size = 0;
    vec->top = 0;

    return 0;
}
```

4.25.2.15 BKNode∗ bkNodeVecGet (BKNodeVec ∗, long)

Definition at line 640 of file primaryPath.c.

References addError(), BKNodeVec_::cont, CRITICAL, and BKNodeVec_::size.

Referenced by bellmanKalaba(), printTopo(), and updateRequest().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (index >= vec->size)
    {
        addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    return vec->cont+index;
}
```

4.25.2.16 int bkNodeVecInit (BKNodeVec ∗, long)

Definition at line 426 of file primaryPath.c.

References addError(), bkConnectVecEnd(), bkConnectVecInit(), BKNODEVEC_INITSIZE, calloc, BKNodeVec_::cont, CRITICAL, free, BKNodeVec_::size, and BKNodeVec_::top.

Referenced by initTopo().

```
{
    BKNode* ptr=NULL;
    long i,j;

    if (vec == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (size == -1)
        size = BKNODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(BKNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<size;i++)
    {
        if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)
        {
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(ptr);
            return -1;
        }
        else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
        {
            bkConnectVecEnd(&ptr[i].inNeighb);
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(ptr);
            return -1;
        }
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return 0;
}
```

4.25.2.17 BKNodeVec∗ bkNodeVecNew (long)

Definition at line 364 of file primaryPath.c.

References addError(), bkConnectVecEnd(), bkConnectVecInit(), BKNODEVEC_INITSIZE, calloc, BKNodeVec_::cont, CRITICAL, free, BKNodeVec_::size, and BKNodeVec_::top.

```
{
    BKNodeVec* vec=NULL;
    BKNode* ptr=NULL;
    long i,j;

    if ((vec = calloc(1,sizeof(BKNodeVec))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return NULL;
    }

    if (size == -1)
        size = BKNODEVEC_INITSIZE;

    if ((ptr = calloc(size,sizeof(BKNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        free(vec);
        return NULL;
    }

    for (i=0;i<size;i++)
    {
        if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)
        {
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(vec);
            free(ptr);
            return NULL;
        }
        else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
        {
            bkConnectVecEnd(&ptr[i].inNeighb);
            for (j=i-1;j>=0;j--)
            {
                bkConnectVecEnd(&ptr[j].inNeighb);
                bkConnectVecEnd(&ptr[j].outNeighb);
            }
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            free(vec);
            free(ptr);
            return NULL;
        }
    }

    vec->size = size;
    vec->top = 0;
    vec->cont = ptr;

    return vec;
}
```

4.25.2.18 int bkNodeVecPopBack (BKNodeVec *, BKNode *)

Definition at line 569 of file primaryPath.c.

References addError(), bkConnectVecCopy(), BKNodeVec_::cont, CRITICAL, BKNode_::inNeighb, BKNode_::neighbInd, BKNode_::nodeId, BKNode_::outNeighb, and BKNodeVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL || val == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top == 0)
    {
        addError(CRITICAL,"Pop on empty stack in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (bkConnectVecCopy(&val->inNeighb,&vec->cont[vec->top-1].inNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    if (bkConnectVecCopy(&val->outNeighb,&vec->cont[vec->top-1].outNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    val->nodeId = vec->cont[vec->top-1].nodeId;
    val->neighbInd = vec->cont[vec->top--].neighbInd;

    return 0;
}
```

4.25.2.19 int bkNodeVecPushBack (BKNodeVec ∗, BKNode ∗)

Definition at line 532 of file primaryPath.c.

References addError(), bkConnectVecCopy(), bkNodeVecResize(), BKNodeVec_::cont, CRITICAL, BKNode_::inNeighb, BKNode_::neighbInd, BKNode_::nodeId, BKNode_::outNeighb, BKNodeVec_::size, and BKNodeVec_::top.

Referenced by fillTopo().

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (vec->top >= vec->size)
    {
        if (bkNodeVecResize(vec,vec->size*2)<0)
        {
            addError(CRITICAL,"Critical lack of memory in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (bkConnectVecCopy(&vec->cont[vec->top].inNeighb,&val->inNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    if (bkConnectVecCopy(&vec->cont[vec->top].outNeighb,&val->outNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }
    vec->cont[vec->top].nodeId = val->nodeId;
    vec->cont[vec->top++].neighbInd = val->neighbInd;

    return 0;
}
```

4.25.2.20 int bkNodeVecResize (BKNodeVec ∗, long)

Definition at line 603 of file primaryPath.c.

References addError(), bkConnectVecInit(), BKNode, BKNodeVec_::cont, CRITICAL, realloc, and BKNodeVec_::size.

Referenced by bkNodeVecPushBack(), and bkNodeVecSet().

```
{
    void *ptr=NULL;
    long i;

    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((ptr = (BKNode*) realloc(vec->cont, newsize*sizeof(BKNode))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    memset(ptr+(vec->size*sizeof(BKNode)) , 0, (newsize-vec->size)*sizeof(BKNode));
    vec->cont = ptr;

    for (i=vec->size;i<newsize;i++)
    {
        if (bkConnectVecInit(&((BKNode*) ptr)[i].inNeighb,-1)<0 ||
            bkConnectVecInit(&((BKNode*) ptr)[i].outNeighb,-1)<0)
        {
            addError(CRITICAL,"Unable to initialize structure in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }
    vec->size = newsize;

    return 0;
}
```

4.25.2.21 int bkNodeVecSet (BKNodeVec *, long, BKNode *)

Definition at line 666 of file primaryPath.c.

References addError(), bkConnectVecCopy(), bkNodeVecResize(), BKNodeVec_::cont, CRITICAL, BKNode_::inNeighb, max, BKNode_::neighbInd, BKNode_::nodeId, BKNode_::outNeighb, BKNodeVec_::size, and BKNodeVec_::top.

```
{
    if (vec == NULL || vec->cont == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index < 0)
    {
        addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (index >= vec->size)
    {
        if (bkNodeVecResize(vec,max(vec->size * 2,index+1))<0)
        {
            addError(CRITICAL,"Unable to resize node vector in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
    }

    if (bkConnectVecCopy(&vec->cont[index].inNeighb,&val->inNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    if (bkConnectVecCopy(&vec->cont[index].outNeighb,&val->outNeighb)<0)
    {
        addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }
    vec->cont[index].nodeId = val->nodeId;
    vec->cont[index].neighbInd = val->neighbInd;
    vec->top=max(vec->top,index+1);

    return 0;
}
```

4.25.2.22 int endTopo (BKTopology ∗)

Definition at line 742 of file primaryPath.c.

References addError(), bkNodeVecEnd(), CRITICAL, longVecEnd(), BKTopology_::nodeInd, and BKTopology_::nodeVec.

Referenced by computePrimaryPath().

```
{
    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    bkNodeVecEnd(&topo->nodeVec);
    longVecEnd(&topo->nodeInd);

    return 0;
}
```

4.25.2.23 int fillTopo (DataBase ∗, LSPRequest ∗, BKTopology ∗)

Definition at line 758 of file primaryPath.c.

References addError(), bkConnectVecEnd(), bkConnectVecInit(), bkConnectVecPushBack(), bkNodeVecPushBack(), calloc, BKConnectVec_::cont, BKNodeVec_::cont, LongVec_::cont, CRITICAL, DBgetLinkState(), DBgetMaxNodeID(), DBgetNbLinks(), DBgetNbNodes(), DBgetNodeInNeighb(), DBgetNodeOutNeighb(), free, BKConnectInfo_::gain, getRequestSrc(), BKConnect_::info, BKNode_::inNeighb, is-

ValidRequestLink(), BKConnect_::linkState, longListEnd, longListInit, longListPopBack, longListPushBack, longVecSet(), NB_OA, BKTopology_::nbLinks, BKTopology_::nbNodes, BKConnect_::neighbId, BKNode_::neighbInd, BKNode_::nodeId, BKTopology_::nodeInd, BKTopology_::nodeVec, BKNode_::outNeighb, BKNodeVec_::top, BKConnectVec_::top, and LongVec_::top.

Referenced by computePrimaryPath().

```
{
    LongList toDoNodes;
    int *activeFlags;
    LongList *tmpNeighb;
    long i,j,nd,src,size;
    BKConnect tmpConn;
    BKNode tmpNode,*nodePtr;


    if (dataBase == NULL || req==NULL || topo==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((src=getRequestSrc(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    size=DBgetMaxNodeID(dataBase)+1;

    if (longListInit(&toDoNodes,size)<0)
    {
        addError(CRITICAL,"Unable to initialize the active nodes list in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((activeFlags = (int*) calloc(size,sizeof(long))) == NULL)
    {
        addError(CRITICAL,"Critical lack of memory in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        return -1;
    }

    memset(&tmpNode,0,sizeof(BKNode));
    if (bkConnectVecInit(&tmpNode.inNeighb,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the temporary node in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        return -1;
    }
    if (bkConnectVecInit(&tmpNode.outNeighb,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize the temporary node in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        bkConnectVecEnd(&tmpNode.inNeighb);
        return -1;
    }

    if (longListPushBack(&toDoNodes,src)<0)
    {
        addError(CRITICAL,"Unable to push back on list of longs in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        bkConnectVecEnd(&tmpNode.inNeighb);
        bkConnectVecEnd(&tmpNode.outNeighb);
        return -1;
    }
    activeFlags[src]=1;
    while (toDoNodes.top>0)
    {
        if (longListPopBack(&toDoNodes,&nd)<0)
        {
            addError(CRITICAL,"Unable to pop back on list of longs in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }

        tmpNode.inNeighb.top=0;
        if ((tmpNeighb=DBgetNodeInNeighb(dataBase,nd))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }
        for (i=0;i<tmpNeighb->top;i++)
        {
            if (activeFlags[tmpNeighb->cont[i]]==2)
            {
                nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
                for (j=0;(j<nodePtr->outNeighb.top) && (nodePtr->outNeighb.cont[j].neighbId!=nd);j++);
                if (j<nodePtr->outNeighb.top)
                {
                    tmpConn.neighbId=tmpNeighb->cont[i];
                    tmpConn.linkState=nodePtr->outNeighb.cont[j].linkState;
                    memset(&tmpConn.info,0,sizeof(BKConnectInfo));
                    memcpy(tmpConn.info.gain,nodePtr->outNeighb.cont[j].info.gain,NB_OA*sizeof(double)
                    if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)
                    {
                        addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                                 __FILE__,__LINE__);
                        longListEnd(&toDoNodes);
                        free(activeFlags);
                        bkConnectVecEnd(&tmpNode.inNeighb);
                        bkConnectVecEnd(&tmpNode.outNeighb);
                        return -1;
                    }
                }
            }
            else
            {
                tmpConn.neighbId=tmpNeighb->cont[i];
                tmpConn.linkState=DBgetLinkState(dataBase,tmpNeighb->cont[i],nd);
                memset(&tmpConn.info,0,sizeof(BKConnectInfo));
                if (isValidRequestLink(dataBase,tmpNeighb->cont[i],nd,
                                       tmpConn.linkState,req,tmpConn.info.gain))
                {
                    if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)
                    {
                        addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                                 __FILE__,__LINE__);
                        longListEnd(&toDoNodes);
                        free(activeFlags);
                        bkConnectVecEnd(&tmpNode.inNeighb);
                        bkConnectVecEnd(&tmpNode.outNeighb);
                        return -1;
                    }
                }
            }
        }

        tmpNode.outNeighb.top=0;
        if ((tmpNeighb=DBgetNodeOutNeighb(dataBase,nd))==NULL)
        {
            addError(CRITICAL,"Unable to get the list of neighbours in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }
        for (i=0;i<tmpNeighb->top;i++)
        {
            if (activeFlags[tmpNeighb->cont[i]]==2)
            {
                nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
                for (j=0;(j<nodePtr->inNeighb.top) && (nodePtr->inNeighb.cont[j].neighbId!=nd);j++);
                if (j<nodePtr->inNeighb.top)
                {
                    tmpConn.neighbId=tmpNeighb->cont[i];
                    tmpConn.linkState=nodePtr->inNeighb.cont[j].linkState;
                    memset(&tmpConn.info,0,sizeof(BKConnectInfo));
                    memcpy(tmpConn.info.gain,nodePtr->inNeighb.cont[j].info.gain,NB_OA*sizeof(double))
                    if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
                    {
                        addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                                 __FILE__,__LINE__);
                        longListEnd(&toDoNodes);
                        free(activeFlags);
                        bkConnectVecEnd(&tmpNode.inNeighb);
                        bkConnectVecEnd(&tmpNode.outNeighb);
                        return -1;
                    }
                }
            }
            else
            {
                tmpConn.neighbId=tmpNeighb->cont[i];
                tmpConn.linkState=DBgetLinkState(dataBase,nd,tmpNeighb->cont[i]);
                memset(&tmpConn.info,0,sizeof(BKConnectInfo));
                if (isValidRequestLink(dataBase,nd,tmpNeighb->cont[i],
                                       tmpConn.linkState,req,tmpConn.info.gain))
                {
                    if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
                    {
                        addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
                                 __FILE__,__LINE__);
                        longListEnd(&toDoNodes);
                        free(activeFlags);
                        bkConnectVecEnd(&tmpNode.inNeighb);
                        bkConnectVecEnd(&tmpNode.outNeighb);
                        return -1;
                    }
                }
            }
            if (activeFlags[tmpNeighb->cont[i]]==0)
            {
                if (longListPushBack(&toDoNodes,tmpNeighb->cont[i])<0)
                {
                    addError(CRITICAL,"Unable to push back on list of longs in %s at line %d",
                             __FILE__,__LINE__);
                    longListEnd(&toDoNodes);
                    free(activeFlags);
                    return -1;
                }
                activeFlags[tmpNeighb->cont[i]]=1;
            }
        }

        tmpNode.nodeId=nd;
        tmpNode.neighbInd=-1;
        if (bkNodeVecPushBack(&topo->nodeVec,&tmpNode)<0)
        {
            addError(CRITICAL,"Unable to push back node in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }

        if (longVecSet(&topo->nodeInd,nd,(topo->nodeVec.top-1))<0)
        {
            addError(CRITICAL,"Unable to set node index in %s at line %d",
                     __FILE__,__LINE__);
            longListEnd(&toDoNodes);
            free(activeFlags);
            bkConnectVecEnd(&tmpNode.inNeighb);
            bkConnectVecEnd(&tmpNode.outNeighb);
            return -1;
        }

        activeFlags[nd]=2;
    }

    if (((topo->nbNodes=DBgetNbNodes(dataBase))<0)||
        ((topo->nbLinks=DBgetNbLinks(dataBase))<0))
    {
        addError(CRITICAL,"Unable to get number of nodes and links in %s at line %d",
                 __FILE__,__LINE__);
        longListEnd(&toDoNodes);
        free(activeFlags);
        bkConnectVecEnd(&tmpNode.inNeighb);
        bkConnectVecEnd(&tmpNode.outNeighb);
        return -1;
    }

    longListEnd(&toDoNodes);
    free(activeFlags);
    bkConnectVecEnd(&tmpNode.inNeighb);
    bkConnectVecEnd(&tmpNode.outNeighb);

    return 0;
}
```

4.25.2.24 int getRequestDst (LSPRequest ∗)

Definition at line 1077 of file primaryPath.c.

References addError(), LongVec_::cont, CRITICAL, LSPRequest_::path, and LongVec_::top.

Referenced by updateRequest().

```
{
    if (req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (req->path.top<2 || req->path.cont[0]<0 ||
        req->path.cont[req->path.top-1]<0)
    {
        addError(CRITICAL,"Bad requested path format in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return req->path.cont[req->path.top-1];
}
```

4.25.2.25 int getRequestSrc (LSPRequest *)

Definition at line 1057 of file primaryPath.c.

References addError(), LongVec_::cont, CRITICAL, LSPRequest_::path, and LongVec_::top.

Referenced by bellmanKalaba(), computePrimaryPath(), fillTopo(), and updateRequest().

```
{
    if (req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (req->path.top<2 || req->path.cont[0]<0 ||
        req->path.cont[req->path.top-1]<0)
    {
        addError(CRITICAL,"Bad requested path format in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return req->path.cont[0];
}
```

4.25.2.26 int initScore (long, BKTopology *)

Definition at line 1402 of file primaryPath.c.

References addError(), DBLinkState_::cap, BKConnectVec_::cont, LongVec_::cont, BKNodeVec_::cont, CRITICAL, damoteConfig, FALSE, BKNode_::inNeighb, BKConnect_::linkState, PrimaryComputationConfig_::loadBal, NB_OA, NB_PREEMPTION, BKTopology_::nodeInd, BKTopology_::nodeVec, DBLinkState_::pbw, DAMOTEConfig_::primaryComputationConfig, BKConnectVec_::top, BKNodeVec_::top, and TRUE.

Referenced by bellmanKalaba().

```
{
    bool process=FALSE;
    long i,j,k,l,top;
    double tmpSum;

    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
    {
        if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
        {
            process=TRUE;
        }
    }

    if (process)
    {
        for (k=0;k<NB_OA;k++)
        {
            topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]=0;
        }
        for (i=0;i<topo->nodeVec.top;i++)
        {
            top=topo->nodeVec.cont[i].inNeighb.top;
            for (j=0;j<top;j++)
            {
                for (k=0;k<NB_OA;k++)
                {
                    tmpSum=0;
                    for (l=0;l<NB_PREEMPTION;l++)
                    {
                        tmpSum+=topo->nodeVec.cont[i].inNeighb.cont[j].linkState->pbw[k][l];
                    }
                    topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]+=tmpSum/
                        topo->nodeVec.cont[i].inNeighb.cont[j].linkState->cap[k];
                }
            }
        }
    }

    return 0;
}
```

4.25.2.27 int initTopo (BKTopology ∗, long)

Definition at line 715 of file primaryPath.c.

References addError(), bkNodeVecEnd(), bkNodeVecInit(), CRITICAL, longVecInit(), BKTopology_::nodeInd, and BKTopology_::nodeVec.

Referenced by computePrimaryPath().

```
{
    if (topo == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (bkNodeVecInit(&topo->nodeVec,-1)<0)
    {
        addError(CRITICAL,"Unable to initialize node vector in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if (longVecInit(&topo->nodeInd,size)<0)
    {
        addError(CRITICAL,"Unable to initialize long vector in %s at line %d",
                 __FILE__,__LINE__);
        bkNodeVecEnd(&topo->nodeVec);
        return -1;
    }

    return 0;
}
```

4.25.2.28 double makeScore (BKTopology ∗, LSPRequest ∗, long, long, BKConnect ∗)

Definition at line 1452 of file primaryPath.c.

References addError(), LSPRequest_::bw, DBLinkState_::cap, LongVec_::cont, BKNodeVec_::cont, CRITICAL, damoteConfig, PrimaryComputationConfig_::delay, BKConnectInfo_::gain, BKConnect_::info, BKConnect_::linkState, PrimaryComputationConfig_::load, PrimaryComputationConfig_::loadBal, makeRerouteScore(), NB_OA, NB_PREEMPTION, BKTopology_::nbLinks, BKTopology_::nodeInd, BKTopology_::nodeVec, DBLinkState_::pbw, DAMOTEConfig_::primaryComputationConfig, PrimaryComputationConfig_::relLoad, PrimaryComputationConfig_::rerouting, PrimaryComputationConfig_::sqLoad, and PrimaryComputationConfig_::sqRelLoad.

Referenced by bellmanKalaba().

```
{
    double score=0,totBW[NB_OA],newSum,rerouteScore=0;
    long i,j;

    if (topo == NULL || connect == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return HUGE_VAL;
    }

    score=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.cost;

    for (i=0;i<NB_OA;i++)
    {
        totBW[i]=0;
        for (j=0;j<NB_PREEMPTION;j++)
        {
            totBW[i]+=connect->linkState->pbw[i][j];
        }

        if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *((totBW[i]+req->bw[i])/connect->linkState->cap[i])
                *((totBW[i]+req->bw[i])/connect->linkState->cap[i]);
            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *(totBW[i]/connect->linkState->cap[i])
                *(totBW[i]/connect->linkState->cap[i]);

            newSum=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkS
            if (__isinf(newSum))
            {
                return HUGE_VAL;
            }

            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *(-1/(double)topo->nbLinks)*newSum*newSum;
            score+=damoteConfig.primaryComputationConfig.loadBal[i]
                *(1/(double)topo->nbLinks)*topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]
                *topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i];
        }
        if (damoteConfig.primaryComputationConfig.load[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.load[i]*req->bw[i];
        }
        if (damoteConfig.primaryComputationConfig.sqLoad[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.sqLoad[i]
                *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i]);
        }
        if (damoteConfig.primaryComputationConfig.relLoad[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.relLoad[i]
                *req->bw[i]/connect->linkState->cap[i];
        }
        if (damoteConfig.primaryComputationConfig.sqRelLoad[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.sqRelLoad[i]
                *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i])
                /(connect->linkState->cap[i]*connect->linkState->cap[i]);
        }
        if (damoteConfig.primaryComputationConfig.delay[i]!=0)
        {
            score+=damoteConfig.primaryComputationConfig.delay[i]
                *((1/(connect->linkState->cap[i]-totBW[i]-req->bw[i]))
                  -(1/(connect->linkState->cap[i]-totBW[i])));
        }
    }

    for (i=0;i<NB_OA;i++)
    {
        if (damoteConfig.primaryComputationConfig.rerouting[i]!=0)
        {
            rerouteScore+=damoteConfig.primaryComputationConfig.rerouting[i]*
                makeRerouteScore(req,connect->info.gain,connect->linkState,i);
        }
    }

    score+=rerouteScore*(score>0?1:0)*score;

    return score;
}
```

4.25.2.29 int noLoop (BKTopology ∗, long, long)

Definition at line 1586 of file primaryPath.c.

References BKConnectVec_::cont, LongVec_::cont, BKNodeVec_::cont, BKNode_::info, BKNode_::inNeighb, BKNode_::neighbInd, BKNodeInfo_::newNeighbInd, BKNode_::nodeId, BKTopology_::nodeInd, and BKTopology_::nodeVec.

Referenced by bellmanKalaba().

```
{
    BKNode* tmpNode;


    tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[src]];
    while (tmpNode->neighbInd!=-1 && tmpNode->nodeId!=dst)
    {
        tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[tmpNode->inNeighb.cont[tmpNode->info.newNeighb
    }

    if (tmpNode->nodeId==dst)
        return 0;

    return 1;
}
```

4.25.2.30 int printTopo (BKTopology ∗)

Definition at line 1015 of file primaryPath.c.

References addError(), bkNodeVecGet(), BKConnectVec_::cont, LongVec_::cont, CRITICAL, BKNode_::inNeighb, BKConnect_::neighbId, BKNode_::neighbInd, BKNode_::nodeId, BKTopology_::nodeInd, BKTopology_::nodeVec, BKNode_::outNeighb, BKConnectVec_::top, and LongVec_::top.

```
{
    BKNode *tmpNode;
    long i,j;

    for (i=0;i<topo->nodeInd.top;i++)
    {
        tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[i]);
        if (tmpNode!=NULL)
        {
            if (i!=tmpNode->nodeId)
            {
                addError(CRITICAL,"Topology unconsistancy in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }

            printf("Node %ld\n--------\n",i);
            printf("Incoming neighboors : \n");

            for (j=0; j<tmpNode->inNeighb.top; j++)
            {
                printf("%ld ", tmpNode->inNeighb.cont[j].neighbId);
            }

            printf("\nOutgoing neighboors : \n");

            for (j=0; j<tmpNode->outNeighb.top; j++)
            {
                printf("%ld ", tmpNode->outNeighb.cont[j].neighbId);
            }
            printf("\n");

            printf("Chosen Neighbour Index: %ld \n",tmpNode->neighbInd);

            printf("\n");
        }
    }

    return 0;
}
```

4.25.2.31 int updateNodeInfoOnElect (BKTopology ∗, LSPRequest ∗, long, long, BKConnect ∗)

Definition at line 1538 of file primaryPath.c.

References addError(), LSPRequest_::bw, DBLinkState_::cap, LongVec_::cont, BKNodeVec_::cont, CRITICAL, damoteConfig, BKConnect_::linkState, PrimaryComputationConfig_::loadBal, NB_OA, BKTopology_::nodeInd, BKTopology_::nodeVec, and DAMOTEConfig_::primaryComputationConfig.

Referenced by bellmanKalaba().

```
{
    long i;

    if (topo == NULL || connect == NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<NB_OA;i++)
    {
        if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
        {
            topo->nodeVec.cont[topo->nodeInd.cont[dst]].info.newSum[i]=
                topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkStat
        }
    }

    return 0;
}
```

4.25.2.32 int updateRequest (BKTopology ∗, LSPRequest ∗)

Definition at line 1097 of file primaryPath.c.

References addError(), bkNodeVecGet(), BKConnectVec_::cont, LongVec_::cont, CRITICAL, getRequestDst(), getRequestSrc(), BKNode_::inNeighb, longListPushBack, BKNode_::neighbInd, BKTopology_::nodeInd, BKTopology_::nodeVec, LSPRequest_::path, and LongVec_::top.

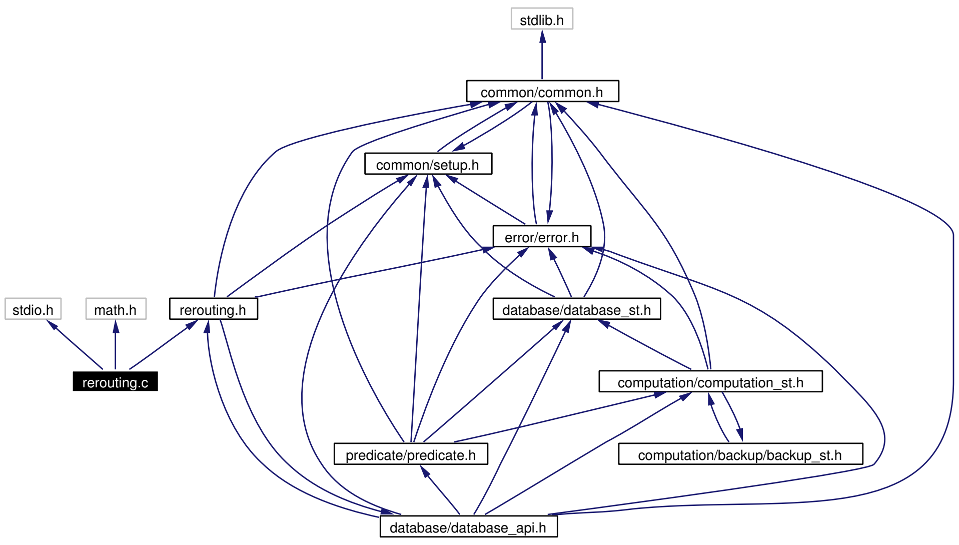
Referenced by computePrimaryPath().

```
{
    BKNode *tmpNode;
    long i,src,dst,nd;

    if (topo == NULL || req==NULL)
    {
        addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((src=getRequestSrc(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    if ((dst=getRequestDst(req))<0)
    {
        addError(CRITICAL,"Unable to get requested source in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }

    req->path.top=0;
    nd=dst;
    if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
    {
        addError(CRITICAL,"Undetermined error in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }
    while (nd!=src)
    {
        if (tmpNode->neighbInd < 0)
        {
            addError(CRITICAL,"Destination unreachable in %s at line %d",
                      __FILE__,__LINE__);
            return -1;
        }
        if (longListPushBack(&req->path,nd)<0)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                      __FILE__,__LINE__);
            return -1;
        }
        nd=tmpNode->inNeighb.cont[tmpNode->neighbInd].neighbId;
        if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
        {
            addError(CRITICAL,"Undetermined error in %s at line %d",
                      __FILE__,__LINE__);
            return -1;
        }
    }
    if (longListPushBack(&req->path,nd)<0)
    {
        addError(CRITICAL,"Undetermined error in %s at line %d",
                  __FILE__,__LINE__);
        return -1;
    }

    for (i=0;i<req->path.top/2;i++)
    {
        nd=req->path.cont[i];
        req->path.cont[i]=req->path.cont[req->path.top-1-i];
        req->path.cont[req->path.top-1-i]=nd;
    }

    return 0;
}
```

4.26 rerouting.c File Reference

```
#include "rerouting.h"
#include <stdio.h>
#include <math.h>
```

Include dependency graph for rerouting.c:



Functions

- int chooseReroutedLSPs (int precedence, DBLinkState ∗state, DBLSPList ∗lspList, double toGain[NB_OA], LongList ∗idList)
- double makeRerouteScore (LSPRequest ∗req, double gain[NB_OA], DBLinkState ∗ls, int oa)

4.26.1 Function Documentation

4.26.1.1 int chooseReroutedLSPs (int *precedence*, DBLinkState ∗ *state*, DBLSPList ∗ *lspList*, double *toGain*[NB_OA], LongList ∗ *idList*)

Definition at line 6 of file rerouting.c.

References addError(), DBLabelSwitchedPath_::bw, LongVec_::cont, DBLSPList_::cont, CRITICAL, DIGIT_PRECISION, DBLabelSwitchedPath_::id, longListPushBack, NB_OA, NB_PREEMPTION, DBLabelSwitchedPath_::precedence, DBLinkState_::rbw, and DBLSPList_::top.

Referenced by DBaddLSP().

```
{
    int i,j,p;
     double need;

     if (idList==NULL || idList->cont==NULL || state==NULL ||
         lspList==NULL || lspList->cont==NULL)
     {
         addError(CRITICAL,"Invalid arguments in %s at line %d",
                      __FILE__,__LINE__);
         return -1;
     }

     for (i=1;i<NB_OA;i++)
     {
         if (toGain[i]>DIGIT_PRECISION)
         {
             addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
                      __FILE__,__LINE__);
             return -1;
         }
         for (j=0;j<NB_PREEMPTION;j++)
         {
             if (state->rbw[i][j]>DIGIT_PRECISION)
             {
                 addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
                          __FILE__,__LINE__);
                 return -1;
             }
         }
     }

     need=toGain[0];
     j=0;
     while (need>DIGIT_PRECISION && j<lspList->top)
     {
         p=lspList->cont[j]->precedence;
         if (p<=precedence)
         {
             addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",
                      __FILE__,__LINE__);
             return -1;
         }

         if (need >= state->rbw[0][p])
         {
             while (j<lspList->top && lspList->cont[j]->precedence==p)
             {
                 longListPushBack(idList,lspList->cont[j]->id);
                 j++;
             }
             need=need-state->rbw[0][p];
             p=p-1;
         }
         else while (need>DIGIT_PRECISION && j<lspList->top)
         {
             if (lspList->cont[j]->bw[0]<=need)
             {
                 longListPushBack(idList,lspList->cont[j]->id);
                 need=need-lspList->cont[j]->bw[0];
                 j++;
             }
             else
             {
                 while (j<lspList->top && lspList->cont[j]->bw[0]>=need && lspList->cont[j]->precedence=
                     j++;
                 longListPushBack(idList,lspList->cont[j-1]->id);
                 need=need-lspList->cont[j-1]->bw[0];
             }
         }
     }

     if (need>DIGIT_PRECISION)
     {
         addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",
                    __FILE__,__LINE__);
            return -1;
        }

        return 0;
 }
```

4.26.1.2 double makeRerouteScore (LSPRequest ∗ *req*, double *gain*[NB_OA], DBLinkState ∗ *ls*, int *oa*)

Definition at line 90 of file rerouting.c.

References addError(), LSPRequest_::bw, CRITICAL, damoteConfig, min, NB_PREEMPTION, LSPRequest_::precedence, DBLinkState_::rbw, DAMOTEConfig_::reroutingConfig, and ReroutingConfig_::scoreCoef.

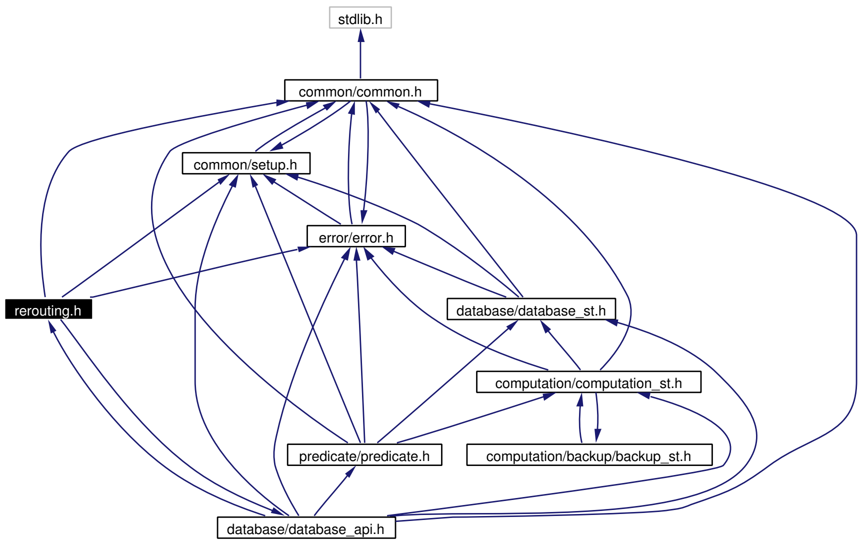
Referenced by makeScore().

```
 {
     int curPrec;
     double bwGained,score=0;

     bwGained=0;
     curPrec=NB_PREEMPTION-1;
     while (bwGained<gain[oa])
     {
         if (curPrec<=req->precedence)
          {
              addError(CRITICAL,"internal error: impossible to realize gain in %s at line %d",
                       __FILE__,__LINE__);
              return HUGE_VAL;
          }
          score=score+damoteConfig.reroutingConfig.scoreCoef[oa][curPrec]*
              min(ls->rbw[oa][curPrec],gain[oa]-bwGained);
          bwGained=bwGained+
              min(ls->rbw[oa][curPrec],gain[oa]-bwGained);
          curPrec--;
      }

      return score/req->bw[oa];
  }
```

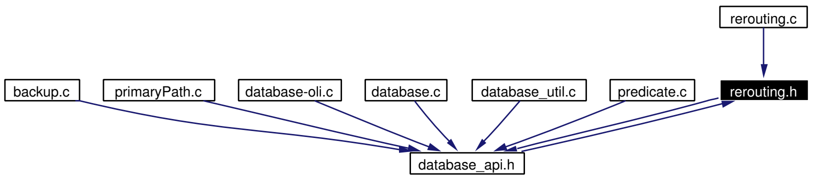
4.27 rerouting.h File Reference

```
#include "common/common.h"
#include "common/setup.h"
#include "error/error.h"
#include "database/database_api.h"
```

Include dependency graph for rerouting.h:



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



Functions

- int chooseReroutedLSPs (int, DBLinkState ∗, DBLSPList ∗, double[NB_OA], LongList ∗)
- double makeRerouteScore (LSPRequest ∗, double[NB_OA], DBLinkState ∗, int)

4.27.1 Function Documentation

4.27.1.1 int chooseReroutedLSPs (int, DBLinkState ∗, DBLSPList ∗, double[NB_OA], LongList ∗)

Definition at line 6 of file rerouting.c.

References addError(), DBLabelSwitchedPath_::bw, DBLSPList_::cont, LongVec_::cont, CRITICAL, DIGIT_PRECISION, DBLabelSwitchedPath_::id, longListPushBack, NB_OA, NB_PREEMPTION, DBLabelSwitchedPath_::precedence, DBLinkState_::rbw, and DBLSPList_::top.

Referenced by DBaddLSP().

```
{
   int i,j,p;
    double need;

    if (idList==NULL || idList->cont==NULL || state==NULL ||
        lspList==NULL || lspList->cont==NULL)
    {
        addError(CRITICAL,"Invalid arguments in %s at line %d",
                     __FILE__,__LINE__);
        return -1;
    }

    for (i=1;i<NB_OA;i++)
    {
        if (toGain[i]>DIGIT_PRECISION)
        {
            addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }
        for (j=0;j<NB_PREEMPTION;j++)
        {
            if (state->rbw[i][j]>DIGIT_PRECISION)
            {
                addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
                         __FILE__,__LINE__);
                return -1;
            }
        }
    }

    need=toGain[0];
    j=0;
    while (need>DIGIT_PRECISION && j<lspList->top)
    {
        p=lspList->cont[j]->precedence;
        if (p<=precedence)
        {
            addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",
                     __FILE__,__LINE__);
            return -1;
        }

        if (need >= state->rbw[0][p])
        {
            while (j<lspList->top && lspList->cont[j]->precedence==p)
            {
                longListPushBack(idList,lspList->cont[j]->id);
                j++;
            }
            need=need-state->rbw[0][p];
            p=p-1;
        }
        else while (need>DIGIT_PRECISION && j<lspList->top)
        {
            if (lspList->cont[j]->bw[0]<=need)
            {
                longListPushBack(idList,lspList->cont[j]->id);
                need=need-lspList->cont[j]->bw[0];
                j++;
            }
            else
            {
                while (j<lspList->top && lspList->cont[j]->bw[0]>=need && lspList->cont[j]->precedence=
                    j++;
                longListPushBack(idList,lspList->cont[j-1]->id);
                need=need-lspList->cont[j-1]->bw[0];
            }
        }
    }

    if (need>DIGIT_PRECISION)
    {
        addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",
                 __FILE__,__LINE__);
        return -1;
    }

    return 0;
}
```

4.27.1.2 double makeRerouteScore (LSPRequest ∗, double[NB_OA], DBLinkState ∗, int)

Definition at line 90 of file rerouting.c.

References addError(), LSPRequest_::bw, CRITICAL, damoteConfig, min, NB_PREEMPTION, LSPRequest_::precedence, DBLinkState_::rbw, DAMOTEConfig_::reroutingConfig, and ReroutingConfig_::scoreCoef.

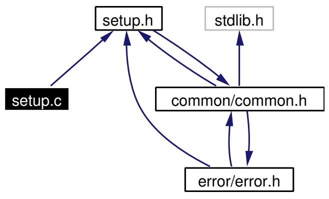
Referenced by makeScore().

```
{
    int curPrec;
    double bwGained,score=0;

    bwGained=0;
    curPrec=NB_PREEMPTION-1;
    while (bwGained<gain[oa])
    {
        if (curPrec<=req->precedence)
        {
            addError(CRITICAL,"internal error: impossible to realize gain in %s at line %d",
                     __FILE__,__LINE__);
            return HUGE_VAL;
        }
        score=score+damoteConfig.reroutingConfig.scoreCoef[oa][curPrec]*
            min(ls->rbw[oa][curPrec],gain[oa]-bwGained);
        bwGained=bwGained+
            min(ls->rbw[oa][curPrec],gain[oa]-bwGained);
        curPrec--;
    }

    return score/req->bw[oa];
}
```

4.28 setup.c File Reference

```
#include "setup.h"
```

Include dependency graph for setup.c:



Variables

- DAMOTEConfig damoteConfig

4.28.1 Variable Documentation

4.28.1.1 DAMOTEConfig damoteConfig

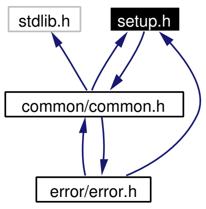
Definition at line 3 of file setup.c.

Referenced by activateNodeInfo(), capacityClause(), initScore(), isValidRequestLink(), makeReroute-Score(), makeScore(), and updateNodeInfoOnElect().

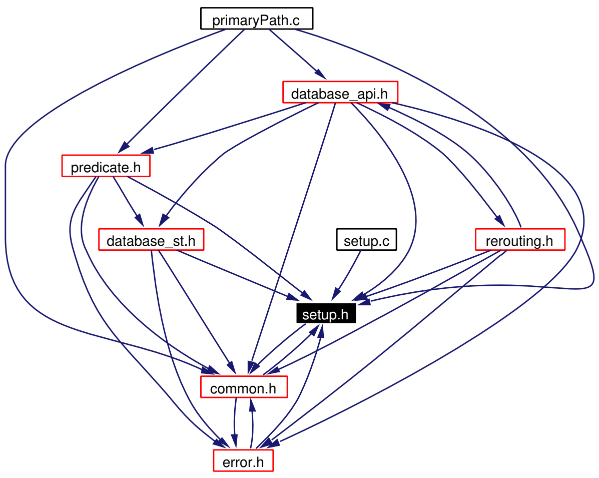
4.29 setup.h File Reference

```
#include "common/common.h"
```

Include dependency graph for setup.h:



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



Data Structures

- struct DAMOTEConfig_
- struct PredicateConfig_
- struct PrimaryComputationConfig_
- struct ReroutingConfig_

Defines

- #define CONTAINER_TEST
- #define LINUX
- #define SIMULATOR
- #define NB_PREEMPTION 1
- #define NB_OA 1
- #define DIGIT_PRECISION 0.0000000001
- #define LSPLIST_INITSIZE 1
- #define LSPREQLIST_INITSIZE 1

- #define NODEVEC_INITSIZE 1
- #define LSPVEC_INITSIZE 1
- #define LINKTAB_INITSIZE 1
- #define ERROR_PROVISION 5
- #define ERRORLIST_INITSIZE 1
- #define ERRORMSG_SIZE 200
- #define LONGVEC_INITSIZE 1
- #define DBLVEC_INITSIZE 1
- #define BKCONNECTVEC_INITSIZE 1
- #define BKNODEVEC_INITSIZE 1

Typedefs

- typedef PrimaryComputationConfig_ PrimaryComputationConfig
- typedef PredicateConfig_ PredicateConfig
- typedef ReroutingConfig_ ReroutingConfig
- typedef DAMOTEConfig_ DAMOTEConfig

Variables

- DAMOTEConfig damoteConfig

4.29.1 Define Documentation

4.29.1.1 #define BKCONNECTVEC_INITSIZE 1

Definition at line 65 of file setup.h.

Referenced by bkConnectVecInit().

4.29.1.2 #define BKNODEVEC_INITSIZE 1

Definition at line 66 of file setup.h.

Referenced by bkNodeVecInit(), and bkNodeVecNew().

4.29.1.3 #define CONTAINER_TEST

Definition at line 12 of file setup.h.

4.29.1.4 #define DBLVEC_INITSIZE 1

Definition at line 63 of file setup.h.

Referenced by dblVecInit(), and dblVecNew().

4.29.1.5 #define DIGIT_PRECISION 0.0000000001

Definition at line 27 of file setup.h.

Referenced by bellmanKalaba(), and chooseReroutedLSPs().

4.29.1.6 #define ERROR_PROVISION 5

Definition at line 58 of file setup.h.

Referenced by addError().

4.29.1.7 #define ERRORLIST_INITSIZE 1

Definition at line 59 of file setup.h.

Referenced by errorInit().

4.29.1.8 #define ERRORMSG_SIZE 200

Definition at line 60 of file setup.h.

Referenced by addError().

4.29.1.9 #define LINKTAB_INITSIZE 1

Definition at line 56 of file setup.h.

Referenced by DBlinkTabInit(), DBlinkTabNew(), and DBnew().

4.29.1.10 #define LINUX

Definition at line 14 of file setup.h.

4.29.1.11 #define LONGVEC_INITSIZE 1

Definition at line 62 of file setup.h.

Referenced by longVecInit(), and longVecNew().

4.29.1.12 #define LSPLIST_INITSIZE 1

Definition at line 51 of file setup.h.

Referenced by DBlspListInit(), and DBlspListNew().

4.29.1.13 #define LSPREQLIST_INITSIZE 1

Definition at line 52 of file setup.h.

Referenced by lspRequestListInit().

4.29.1.14 #define LSPVEC_INITSIZE 1

Definition at line 55 of file setup.h.

Referenced by DBlspVecInit(), and DBlspVecNew().

4.29.1.15 #define NB_OA 1

Definition at line 25 of file setup.h.

Referenced by activateNodeInfo(), capacityClause(), chooseReroutedLSPs(), computeBackup(), computeCost(), computeRBW(), DBaddLSP(), DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), DBlinkStateNew(), DBlspCopy(), DBlspInit(), DBprintLink(), evalLS(), fillTopo(), initScore(), isValidLSPLink(), lspRequestInit(), lspRequestNew(), makeScore(), updateLS(), and updateNodeInfoOnElect().

4.29.1.16 #define NB_PREEMPTION 1

Definition at line 24 of file setup.h.

Referenced by capacityClause(), chooseReroutedLSPs(), computeBackup(), computeCost(), computeRBW(), DBlinkStateCopy(), DBlinkStateDestroy(), DBlinkStateEnd(), DBlinkStateInit(), DBlinkStateNew(), DBprintLink(), initScore(), makeRerouteScore(), and makeScore().

4.29.1.17 #define NODEVEC_INITSIZE 1

Definition at line 54 of file setup.h.

Referenced by DBnodeVecInit(), and DBnodeVecNew().

4.29.1.18 #define SIMULATOR

Definition at line 22 of file setup.h.

4.29.2 Typedef Documentation

4.29.2.1 typedef struct DAMOTEConfig_ DAMOTEConfig

4.29.2.2 typedef struct PredicateConfig_ PredicateConfig

4.29.2.3 typedef struct PrimaryComputationConfig_ PrimaryComputationConfig

4.29.2.4 typedef struct ReroutingConfig_ ReroutingConfig

4.29.3 Variable Documentation

4.29.3.1 DAMOTEConfig damoteConfig

Definition at line 106 of file setup.h.

Referenced by activateNodeInfo(), capacityClause(), initScore(), isValidRequestLink(), makeRerouteScore(), makeScore(), and updateNodeInfoOnElect().

Index